A FRAMEWORK FOR PROMOTION ANALYSIS IN MULTI-DIMENSIONAL SPACE

BY

TIANYI WU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Professor Jiawei Han, Chair
Professor Marianne Winslett
Associate Professor ChengXiang Zhai
Venkatesh Ganti, Ph.D., Google Inc.

Abstract

Promotion is one of the most important elements in marketing. It is often desirable to find merit in an object (*e.g.*, product, person, organization, or other business entity) and promote it in an appropriate community confidently. In this thesis, we motivate and discuss a novel class of data mining problems, called *promotion analysis*, for promoting a given object in a multi-dimensional space by leveraging object ranking information. The key observation is that most objects may not be highly ranked in the global space, where all objects are compared by all aspects; in contrast, there often exist interesting and meaningful local spaces in which the given object becomes prominent. Therefore, our general goal is to break down the data space and discover the most interesting local spaces in an effective and efficient way. We formally present the promotion analysis problem and formulate its variants and related notions.

The promotion analysis problem is highly practical and useful in a wide spectrum of decision support applications. Typical application examples include merit discovery, product positioning and customer targeting, object profiling and summarization, identification of interesting features, and explorative search of objects. In fact, these applications are not new as they have been extensively studied and practiced in the marketing field. While existing commercial database and business intelligence systems can well support the functionality of retrieving the most highly ranked objects in some local space, there exists no multidimensional ranking analysis study for promotional purposes. Supporting effective and efficient online promotion analysis, nevertheless, presents many technical challenges, such as the spurious promotion problem, explosion of search space, and the high complexity of aggregation. Toward this end, we systematically study the problem and develop a general, principled promotion analysis framework. In terms of the search space, both subspaces formed on categorical dimensions and regions formed on continuous dimensions are examined. In terms of the object domain, both uniform object collection and multidimensional object space are

studied. Moreover, we propose a unified query model to accommodate various scoring functions and redundancy-aware semantics. We also develop a statistical method to avoid spurious promotion results. For efficient query processing with a desirable balance between online and offline costs, we investigate exact algorithms as well as approximate algorithms with probabilistic guarantee.

The promotion analysis framework not only provides an integrated solution for decision support applications, but also opens up new horizons for future research in other areas like information network analysis, text mining, and probabilistic data management.

To Jingjing and my parents.

Acknowledgments

First and foremost, I would like to express my deepest and most sincere appreciation to my advisor Professor Jiawei Han, who has been providing continuous guidance as well as persistent support throughout my Ph.D. study. His vision and direction about research have greatly inspired me and broadened my scope of knowledge. His caring and encouraging personality brought me through many difficulties. I am tremendously thankful for the help he offered every step of the way: identifying interesting problems, developing rigorous work, and evaluating and selling ideas. In addition to research, one of the best lessons I have received from Professor Han is the way he nurtures originality, integrity, and independence, which has fundamentally changed my mindset, and I am truly grateful for that.

I would also like to show my deep gratitude to other doctoral committee members, Professor Marianne Winslett, Professor ChengXiang Zhai, and Dr. Venkatesh Ganti for offering their critical suggestions and insightful perspectives on the thesis. Their suggestions help me improve it in a more accurate and comprehensive way.

Many thanks to my mentors at Microsoft Research, Dr. Kaushik Chakrabarti, Dr. Li-wei He, and Dr. Hung-chih Yang. Under their hand-in-hand guidance, I have gained a lot of valuable experiences and learned from their way of thinking. Also, I would like to thank Professor Yuguo Chen, Professor Cuiping Li, Professor Jianlin Feng, Professor Guozhu Dong, and Dr. Kaizhi Tang for their constructive discussions and comments.

As a member of the Database and Information System (DAIS) lab, I thank all professors, colleagues, and friends, who have given me a lot of help. I especially thank Dong Xin, Xiaolei Li, Hector Gonzalez, Qiaozhu Mei, Yizhou Sun, Zhenhui Li, Ricardo Redder, Jacob Lee, and Yintao Yu. Also thanks to my friends and colleagues in DAIS: Sruthi Bandhakavi, Dustin Bortner, Deng Cai, Chen Chen, Hong Cheng, Marina Danilevsky, Bolin Ding, Jing Gao, Manish Gupta, Ming Ji,

Xin Jin, Hyung Sul Kim, Sangkyum Kim, Min-Soo Kim, Meghana Kshirsagar, Rui Li, Cindy Xide Lin, Chao Liu, Yue Lu, Zheng Shao, Lu An Tang, Arash Termehchy, Joana Trindade, Chi Wang, Xuanhui Wang, Tim Weninger, Xifeng Yan, Xiaoxin Yin, Zhijun Yin, Xiao Yu, Duo Zhang, Bo Zhao, Peixiang Zhao, and Feida Zhu.

All research work would have not been possible without the financial support and various kinds of assistance from many institutions and agencies including the Computer Science Department at the University of Illinois at Urbana-Champaign, National Science Foundation, Air Force Office of Scientific Research, Army Research Lab, and Intelligent Automation, Inc..

Last but not least, my heartfelt appreciation goes to my wife Jingjing, my parents, and my whole family. I genuinely feel blessed to be a part of the family and I am deeply indebted for their love and inspiration.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Promotion has been playing a key role in marketing. It is always desirable to identify the competitive strengths of a product and promote it on the market. Existing decision support systems, equipped with online analytical processing (OLAP) and multidimensional analysis engines, can accommodate marketing and business intelligence applications by helping users conduct explorative analysis and gain insights from data. They enable various aggregation techniques as well as basic operations like roll-up and drill-down. However, none of these systems is able to help users navigate and analyze data for promotional purposes; there is still a need to systematically support a function that automatically discovers interesting and meaningful information for a given object so that it can be promoted confidently.

In this thesis, we propose and study a novel data mining problem, called *promotion analysis through ranking*, that will be useful for many decision support applications. In a nutshell, we exploit a common fact that ranking information, in particular top ranks, of a target object (*e.g.*, product or person) can serve as effective means for promotional purposes. For example, "Fortune 500" could deliver a positive image of an enterprise to its customers, and the fact that a book seller has the third largest readership among college students can also guide marketers' strategies. Because this type of interesting ranks can be clearly valuable to promote a given object in a number of practical scenarios, our goal here is to leverage the ranking information for object promotion. Unfortunately, observe that in many cases such "highly ranked" information would be simply impossible to obtain for a majority of objects since they may not be prominent globally; that is, when comparing to *all* their competitor objects in regard to *all* aspects. Thus, toward our goal, ranking analysis in subspaces or local regions in the multidimensional data space must be carried out to mine useful knowledge for promotion. Let us first consider several examples.

Example 1 (PRODUCT PROMOTION) *A book retailer manager intends to promote their brand. Unfortunately, in terms of book sales, they are ranked lower than 30-th among all retailers. However, when breaking down the market into segments, such as* Year*,* Category*, and* Readership*, she finds out that they are in fact* the top-1 bookseller in the {Readership = College Students, Category = Science and Technology} segment*. This piece of information can be then used for making advertisement and allocating marketing resources to seek profits.*

Example 2 (PERSON PROMOTION) *An NBA manager would like to promote Michael Jordan as a superstar. Having checked the statistics, he realizes that Jordan is only ranked as the 3rd all-time leading scorer. However, further analysis suggests more exciting results:* Jordan is the top scorer in the guard position, the top scorer on the Chicago Bulls team, as well as 11 individual years' scoring champion.

Example 3 (PROMOTION IN CONTINUOUS SPACE) *To promote a hybrid car model, a data analyst may discover an interesting promotion region to be* {Year = 2008 $\sim$ 2010; Price = \$15K $\sim$ \$20K}*, in which it receives the highest customer rating among all cars. Another example region could be like* {CustomerIncome = \$20K $\sim$ \$30K}*, where the car model is in the top-3 by rating. These regions, in contrast to the global region in which the car model has a much lower rank, provide more concrete and insightful information for product positioning.*

The above examples illustrate typical applications of the promotion analysis function. The strategy adopted here is to break down the data space into local spaces so that a globally low-ranked object becomes prominent in some interesting subspaces, which can be then used for promotion. In fact, this strategy is not new as it has been extensively studied and practiced in various marketing applications [49, 37]. Unfortunately, while existing commercial database and data warehouse systems can well support the functionality of "retrieving the top-ranked objects in some subspace", there exists no subspace ranking analysis method for promotional purposes. It would be prohibitively difficult for users to navigate large data sets because of two reasons. First, there is potentially an exponential number of local spaces with respect to the total number of dimensions, so it is impossible to exhaustively enumerate each one. Second, the aggregation and ranking of objects often incur high computational cost, making the overall computation process inefficient.

Figure 1.1: A product web page snapshot from Amazon.com.

What happens to users of a conventional database or data warehouse system is that they would need to go through a trial-and-error process to manually search for interesting local spaces, meaning that they have to heavily rely on their prior knowledge. The results obtained in this way, however, could be rather incomplete or even misleading.

Figure 1.1 displays a snapshot of a product web page from Amazon.com, where the sales ranking information of the product is highlighted in the red rectangle. We can see that the product's ranks in two local spaces, namely "Electronics" (6996th) and "Digital SLRs" (80th) are shown. However, these ranks may not be the best to promote the product or impress customers as they are more inclined to choose the top-ranked ones. This motivates us to thoroughly explore the ranking space and identify the most interesting ranked results. In the thesis work, we will systematically study the following problem, which, to the best of our knowledge, has not been studied before.

Definition 1 (The Promotion Analysis Problem) *Given a target object of interest, our goal is to discover its top-k interesting local spaces for promotion.*

To formulate this problem, a data context and proper query semantics must be defined. We use the multidimensional data model to accommodate typical decision support data. The multidimensional model consists of three types of dimensions, *subspace dimensions*, on which local spaces or regions are defined, *object dimensions*, which contain a collection of objects and object

Location	Year	Product	Feature	Sales	Rating
L_1	2010	P_1	f_1	120	3.7
L_3	2009	P_1	f_1	150	3.0
L_2	2010	P_1	f_2	200	4.0
L_2	2010	P_2	f_2	220	4.6
L_3	2010	P_2	f_2	160	5.0
L_1	2009	P_2	f_2	180	4.3
L_2	2010	P_2	f_3	230	3.5
...	...	...	...	...	...

Table 1.1: An example multidimensional data set.

groups/features, and *score dimensions*, from which object rankings are derived. An example multidimensional data set is displayed in Table 1.1. In the table, the categorical dimension *Location* and the continuous, ranged dimension *Year* are the subspace dimensions. An example local space could be "$\{L_1, 2009 \sim 2010\}$". *Product* and *Feature* are the object dimensions, where each product may have multiple features. Finally, *Sales* and *Rating* are two score dimensions; objects can be ordered ascendingly or descendingly according to these criteria. In different contexts, two types of object rankings can be distinguished. (i) Aggregate score-based: the computation of an object's rank in any local space involves dynamic score aggregation; (ii) Constant score-based: each object's score is constant across all local spaces, so objects' relative ranking will be fixed in any space. We focus on the former case, which is more general.

In order to quantify the promotional value of a local space, we need to have an interestingness measure. The interestingness measure in principle should consider the following major factors. First and foremost, the target object's rank is a decisive component to judge a local space because our promotion analysis focuses on the ranking aspect of the object. Second, we consider the rank-independent significance of the local spaces themselves since not all spaces are equal. Users may assign ad-hoc weights to measure the interestingness. Third, more complex query semantics may be adopted, such as ensuring that the top-k spaces discovered do not contain any redundancy. We will elaborate on different problem formulations and variants in the subsequent chapters.

The promotion query function can benefit many applications. In business intelligence applications like marketing, it is able to assist data analysts to quickly locate and understand which spaces are the most likely to promote some specified product or business object among a myriad of candidate spaces, and then they can leverage the query results to serve decision making purposes.

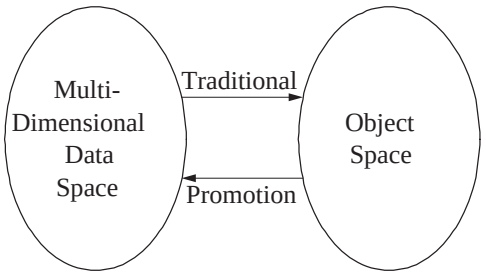


Figure 1.2: The search space of the promotion analysis problem.

Generally, the promotion analysis function has the following uses.

- The promotion analysis results can be directly used to find merit and identify competitive strengths of a target object, raising its image and profile;
- Helps discover the right market segment for resource allocation, positioning, and targeting (*e.g.*, a bestselling product in a particular product category or customer space). Such market segments can be defined over both categorical or continuous ranged dimensions;
- Delivers more specific and informative results (*e.g.*, "a university ranked top-3 in biomedical research" could be more informative and useful than a general report that "it is ranked top-15 among all universities");
- Summarizes interesting object groups and features (*e.g.*, one may find that the PC retailer Dell's best product feature in comparison to all other retailers to be "small business desktop", or that certain product packages are more liked by customers than others).

Technically, the promotion function can also help detect anomaly or extreme aggregates across different spaces. Although the need for such a data mining functionality is ubiquitous, no previous work is able to address this problem. Figure 1.2 illustrates the distinction between this problem and previous ones. Conventional ranked query problems focus on retrieving the highly ranked objects in a particular local space specified by the user, and thus the search space is the object space, whereas in the promotion analysis problem, we focus on discovering the most interesting local spaces, and thus the search space is the multidimensional data space. Despite the similarity with previous studies in terms of modeling the data and object spaces, the promotion analysis

problem presents significant new research challenges in both query semantics and computation as follows:

- (Search Space Explosion) In a d-dimensional data set, there is potentially an exponential number of local spaces. A naive approach that enumerates all local spaces and computes the rank in each one would be prohibitively expensive;
- (Non-Monotonicity) To produce object rankings, an aggregation measure must be used. This measure, however, may or may not be monotonic. For example, the *Average* measure does not satisfy the monotonicity property (*i.e.*, the measure value in a parent space may or may not be larger than that in its child spaces). This non-monotonicity property of the aggregation measure prevents us from utilizing existing aggregate computation methods;
- (Holistic Measure) The rank measure is holistic [35], which means that computing object ranking incurs high aggregation cost because scores must be aggregated for *all* objects in order to derive the query object's rank;
- (Spurious Promotion) Sometimes, a seemingly interesting local space where an object is highly ranked might be caused by random perturbation and is therefore meaningless for promotion. Such spurious promotion needs to be avoided from query results;
- (Query Semantics) Different spaces may not be equally interesting; specifically, they may have (i) dramatically different sizes, and (ii) containment relationships or overlaps that lead to redundancy in query results. To this end, the semantics of the model must incorporate rank-independent weights and make the results discriminative.

We will investigate the principles and methodologies for the promotion analysis problem. Our study will be twofold: (i) to develop effective query models for mining interesting and meaningful patterns, and conduct case study to validate the mined results; and (ii) to develop efficient, scalable algorithms for answering promotion queries.

Specifically, we will focus on the following research issues. In Chapter 3, we study the promotion analysis problem in the multi-dimensional, categorical space, where the subspace dimensions considered in the problem contain only discrete values [88]. A unified interestingness measure is

proposed to model various scenarios such as simple ranking and percentile ranking. Both online and offline techniques are developed and integrated into a recursive aggregation framework for query optimization. Moreover, we also introduce the spurious promotion problem: a local space in which the target object is highly ranked may not be meaningful due to that this local space contains at least one spurious dimension. Since spurious dimensions have no correlation with the object rankings, they can be detected and removed using statistical methods. We conduct a case study on two real data sets to demonstrate the effectiveness of the results and the efficiency of our proposed methods.

In Chapter 4, the region-based promotion query problem over continuous, ranged dimensions is examined [86]. The query model incorporates ad-hoc weights over different regions and the redundancy-aware semantics. This problem introduces several major new challenges due to a significantly larger the search space (*i.e.*, the number of regions) and a more flexible query model in comparison to the previous problem. To solve it, a region-based promotion cube framework is developed using a solid theoretical analysis. We develop a structure to yield the provably optimal pruning power and a relaxed structure to further optimize the storage overhead. Experiments show that the methods can achieve a much better tradeoff between the offline and online costs.

In Chapter 5, we further extend the data space to handle multidimensional object relationships. Instead of treating objects as a single-typed, homogeneous collection, objects are allowed to form hierarchical or interrelated relationships. Therefore, objects can be ranked at different levels through the comparison of different features or groupings. We formulate the promotion analysis problem over such a multidimensional object space and study approximate query processing techniques with precision guarantee. A regression-based structure and a probabilistic pruning algorithm are proposed. Our experiments show that these techniques significantly outperform exact techniques while sacrificing very little accuracy in query results.

The remainder of the thesis is organized as follows. Chapter 2 provides an overview of the related work. Chapter 6 summarizes our study and outlines the directions for future work.

Chapter 2

Related Work

In this chapter, we review the related work of the promotion analysis problem. We first overview a collection of data mining methods driven by various marketing applications. Then, we survey the literature on online analytical processing (OLAP) and decision support, which are closely related to our problem. Also, we will discuss existing work on ranked query processing, which present similar technical challenges.

2.1 Data Mining for Marketing

Promotion is one of the four key ingredients (*i.e.*, four P's) in marketing: Product, Price, Place, and Promotion [49], which serves for the purposes of developing brand, building awareness, and obtaining and retaining customers. It has been recognized that data mining methodologies play an important role in promotion as well as other marketing applications. Because businesses are accumulating increasingly large amounts of data, there comes a strong need to analyze the data, discover hidden patterns and useful insights from it, and generate actionable plans and profit. Systematic research and development of data mining methodologies will help achieve this goal by transferring such large amounts of data into knowledge and actionable plans [7, 75, 4, 72, 37]. Many methods have been developed and shown to be beneficial to a wide variety of applications, including sales and customer support [7], viral marketing [72], marketing in social networks [61, 40, 34], and product recommendation and positioning [53, 50, 53, 66, 82]. These methods have also employed techniques from database systems, machine learning, information retrieval, human-computer interaction, and social sciences.

In [48], the authors study the utility of data mining operations like association rule mining and clustering for decision making. It presents a general theoretical framework to model the usefulness

of the operations from a practical point of view. In the area of online search-based advertising, new recommendation models have been proposed [73, 10] to optimize keyword selection and bid strategies, whose objective is to maximize the profit from advertising. In the database community, the dominant relationship analysis problem is proposed [53]. Given a large data set, this work aims at modeling the relationships between product attributes and potential customers in order to position the product appropriately. In addition, [82] discusses the problem of creating competitive products among a large number of candidate products. Its objective is to ensure that a newly created product be no worse than any existing product in terms of a set of attributes.

Another fruitful area of data mining research that marketing applications have benefited from is the association rule analysis problem [37, 50, 83, 84]. Given a market basket data set such as the Walmart transaction data, the problem aims at discovering interesting associations rules between sets of items. Dozens of measures, such as support and confidence, have been proposed [37, 83, 84] to judge the interestingness of association rules. For example, a data analyst may find that an item milk is highly associated with another item bread in that they appear together frequently; subsequently, marketing activities like promotional pricing and product placements can be performed to seek profit.

The promotion analysis problem is related to the above problems as far as the high-level marketing goals are concerned. However, promotion analysis cannot be replaced by any existing data mining method since its main objective is to leverage highly ranked results for promotion, which has never been studied before.

2.2 Online Analytical Processing (OLAP) and Decision Support

The promotion analysis problem is closely related to online analytical processing (OLAP) and decision support systems, which aim at supporting flexible and efficient multidimensional analysis [17, 37]. In a typical decision support system, the multidimensional or data cube model is used [35], which consists of a collection of fact dimensions such as location, year, and other product attributes. Also, there could be one or more numerical measure dimensions, such as product price, sales, and user ratings. The metadata of the multidimensional data model is often maintained using a star schema or a snowflake schema. This model, along with a set of primitive OLAP operations

(*e.g.*, roll-up, drill-down, slice, and dice) allows users to aggregate the data and generate summary reports in a convenient way. For example, a data analyst may first check the aggregated sales data of the recent year and then drill down to selectively explore certain locations and months she is interested in. Since the OLAP approach is integral to decision support systems, it is shown to be a powerful tool for business intelligence applications.

To enable efficient OLAP operations, aggregates are often precomputed across different cube (or group-by) spaces. Such precomputation can greatly speed up online query processing, thereby facilitating interactive data exploration. Note that different aggregate measures have different computational complexities; specifically, three categories of measures can be distinguished. First, a *distributive* measure (*e.g.*, SUM) can be computed in a bottom-up fashion such that the aggregate of a cube cell can be derived from its children cells. Second, an *algebraic* measure (*e.g.*, $Average$) can be computed using some other measures (*e.g.*, SUM and $COUNT$). Third, a *holistic* measure (*e.g.*, $Rank$) of a cell cannot be derived from its children cells using a constant amount of space. It is often much more costly to compute a holistic measure than a distributive or algebraic measure. Since fully materialize all aggregates would be prohibitively expensive, many algorithms have been proposed to further optimize the time- or space-efficiency of multidimensional analysis. For instance, [39] proposes an approximation method to choose the best set of materialized views for maximizing online query processing efficiency, and [41] aims at optimizing range aggregate queries. [56] further proposes to use inverted indices for supporting high-dimensional OLAP operations.

Instead of computing exact answers, approximate OLAP results are often acceptable for exploration purposes. Among the approximate techniques, [80, 79] propose to reduce the storage overhead using wavelets, whereas [74] employs the Gaussian mixture model to approximate continuous dimensions.

A variant of the OLAP cube computation problem is called the iceberg queries [30]. By introducing an iceberg threshold, sparse aggregates will be filtered and only those passing the threshold will be computed so that the CPU cost and storage space will be greatly reduced. Note that finding frequent items [14] can be considered as a closely related problem. Many algorithms are developed for computing iceberg cubes. In [8], a recursive bottom-up aggregation method is developed to compute an iceberg cube, whereas [38] studies efficient methods for computing complex measures.

The key thrust of these methods lies in the *monotonicity* property of a measure. For example, for a measure like *SUM*, the measure of a cell must be no smaller than that of its children cells. Therefore, for any aggregated cell that cannot pass the iceberg threshold, all of its children cells can be pruned.

Another way to approximate the aggregated data is to compute and maintain quantiles. A quantile is simply a set of values (or order statistics) taken at evenly spaced intervals from a distribution. In OLAP, quantiles can be used to provide good approximation of the distribution of aggregates while using very small storage space. The quantile computation and maintenance problem has been thoroughly studied in the database field. Many researchers investigate time- and space-efficient algorithms for exact or approximate quantile computation [62, 63, 36, 32, 21, 77]. For the promotion analysis problem, we also develop quantile-like data structures for efficient online query processing. However, the use of our data structures is not for approximating the distribution, but for bounding the ranks of a query object. We will model the utility of the data structures for promotional purposes and construct optimal structures; these objectives have not been considered before.

Besides supporting basic aggregate measures, various data mining functions can be integrated with OLAP to support other applications. For example, [69] studies predictive model construction in the cube space and [55] investigates the anomaly detection problem for multidimensional time-series data. For regression analysis, efficient computation methods are proposed in [19]. Moreover, OLAP on RFID flow data and graph data is discussed in [33, 18]. [20, 59] aim at supporting OLAP operations for sequence databases and [93] provides operations for search engine logs. The multidimensional analysis has also been extended to the unstructured data domain. Efficiently computing the information retrieval measures like TF-IDF for text databases is studied in [58]. [92] proposes an algorithm to efficiently construct topic models, and [25] develops methods for answering keyword search queries for aggregated documents. Finally, [27] devises a technique for mining anomaly cells that have significant measure deviation from its surrounding cells.

Despite that many approaches have been proposed so far, none of them is able to support promotion analysis. Our study complements the existing OLAP functions in two ways: (i) the concept of discovering interesting subspaces for promotion is new, and (ii) the techniques developed

in the thesis enable users to conduct promotion analysis efficiently and flexibly.

2.3 Ranked Query Processing

Ranked or top-k query processing is yet another research area that is closely related to the promotion analysis problem due to the technical similarity. Ranked query processing has been extensively studied in Web search, database systems, and other fields. Among all techniques developed, Fagin's threshold algorithm (or TA) [29] is one of the most important. Given m lists of objects, each list being sorted according to some score attribute, TA is able to efficiently compute the top-k objects based on some monotone aggregate function that combines the m scores. This algorithm has uses in many applications like multimedia databases and information retrieval. For example, TA can be used to compute the top-k documents with the highest relevance scores by combining multiple feature lists, where each list contains all documents sorted according to the feature.

In relational database systems, the ranked query model augments the traditional boolean query model by enabling ranking of tuples or aggregates. Users are allowed to express their preference through some ranking function. As a result, the ranked query results are ordered descendingly in terms of a user's interest. This would also solve the too-many-answer problem as many SQL queries would return too many tuples that are difficult for the user to navigate. Numerous query models and techniques have been developed for effective and efficient ranked query processing [13, 28, 64, 45, 5, 16, 23]. [46] presents a survey and categorization of different techniques. While ranked queries are somewhat similar to iceberg queries [30] in that both types of queries can limit the cardinality of results to users and eliminate less interesting tuples, they are intrinsically different, because ranked queries do not a require hard threshold. Therefore, ranked queries are more desirable for the applications in which users do not have prior knowledge about what iceberg threshold should be specified in the first place.

The ranked query problem has been notably discussed in the context of multidimensional data and aggregate queries [90, 89, 85, 87]. The first study on integrating ranking and multidimensional analysis is the ranking-cube approach [90, 89], which aims at efficiently supporting top-k queries with multidimensional selections. For example, a user may want to find, according to her preference, the top-k cars matching a given boolean condition. To solve this problem, the ranking-cube

Query model	*Object*	*Rank*
Conventional	Output	Limits the output cardinality
Promotion	Input	Interestingness measure

Table 2.1: A comparison of traditional query models and the promotion query model.

approach partitions the measure space and precomputes inverted indices for efficient online query processing. Further, [87] devises an algorithm for ranked aggregate query processing. A partial materialization strategy is proposed and an early stopping condition is established for a set of aggregate measures.

The ranked query problem appears not only in relational tables with numerical measures, but also many other scenarios. For example, efficient top-k techniques have been developed for keyword search in relational databases with text fields [43, 12, 60], where tuples are ranked by relevance scoring functions. Researchers have also studied ranking models on the graph data [91] as well as probabilistic data management [78, 71, 76, 44, 54].

The promotion query model can be distinguished from the previous ranked query models as the target object in our problem context is specified upfront as a user *input* (*e.g.*, a product to be promoted) as opposed to be an *output* query result (*e.g.*, a highly ranked product in some given space). Another critical difference is that we use *rank* as a measure for promotion, while in previous studies, the rank is used for returning a digestible set of relevant answers to the user. A summarization of the differences is displayed in Table 2.1.

Toward the problem of finding top-k attributes, [24] investigates how to select the most useful attributes to explain ranked tuples. [65] aims at finding the best attributes to maximize for a given query workload the number of queries which can retrieve a given tuple. Moreover, [81] tries to find numerical parameter spaces in which a given object is in the top-k. Unlike most ranked query models, these problems try to identify interesting attributes or queries rather than tuples. Nevertheless, they are different from our problem in several ways. First, their main objectives are not to explore ranking for promotional purposes, and their search spaces focus on the attribute-level or the score space rather than the multidimensional space formed by categorical or continuous dimension values. Moreover, aggregation is not considered in these problems, whereas it is used in our setting to produce object rankings.

2.4 Others

There are several other problem spaces that are related to promotion analysis. As illustrated in Figure 1.2, promotion queries can be considered, at a high level, as a reverse query processing problem. There are various formulations of reverse query processing in other applications, such as nearest neighbor search, skyline queries, and database testing [6, 9, 57]. In the skyline query problem setting, there are also studies that try to find interesting subspaces [11, 68]. However, the methods developed for those problems are not applicable to promotion analysis because they do not deal with ranking or aggregation.

Technically, this thesis work has employed statistical methods for result validation and probabilistic pruning [52, 51]. Part of the techniques shares similar objectives as some previous approximation algorithms [42, 47]. For example, [15, 26, 31] propose polynomial approximation algorithms for clustering data points so as to minimize the sum of cluster diameters or radii. However, these approximation methods are of theoretical interests only, and they are not efficient to be applied to database query processing nor scalable to deal with large data sets.

Chapter 3

Promotion Analysis in Multi-Dimensional Space

3.1 Introduction

In this chapter, we study the basic promotion query problem over categorical dimensions. The problem can be stated as follows: given a *target object* such as a product or a person, our goal is to discover its top-k *promotive* subspaces. Here k is a user-specified non-negative integer, and subspaces are defined over categorical dimensions (*e.g.*, a subspace for a book sales database could be like {*Readership=College Student, Category=Science and Technology*}). To address this problem, a new notion called *promotiveness* needs to be formulated, which is an interestingness measure that gauges how well a subspace can promote the target object. Intuitively, that the target object is highly ranked in a subspace suggests that the subspace is *promotive*. The promotion query returns the target object's k most promotive subspaces, which may have different uses depending on the application scenario. First, the promotion query results can be directly used to find merit and competitive strengths of the target object, thereby enhancing its image and profile. Second, the promotion query enables data analysts to conveniently search for the right market segments for further marketing activities. For example, if a book is highly ranked among college students, the marketer may want to allocate more marketing resources or launch campaigns a on the college segment. Third, high rankings in promotive subspaces can be more meaningful and informative to an information seeker in many cases. For example, Forbes and U.S. News and World Report regularly publish ranked results of businesses, universities, and other organizations or persons not only in general, but also in various subfields (*e.g.*, universities are ranked by undergraduate vs. graduate education, and businesses are ranked by their category or size); such subspace rankings can be more interesting and telling than a general, overall ranking.

Location	Year	Object	Score
NY	2008	T_1	0.5
WA	2008	T_1	0.8
WA	2007	T_2	1.0
WA	2008	T_2	1.0
NY	2007	T_3	0.3
WA	2007	T_3	0.6
WA	2008	T_3	0.7

Figure 3.1: Sample multidimensional data.

Subspace	Rank	ObjCount
{*}	3rd	3
{NY}	1st	2
{WA}	3rd	3
{2008}	1st	3
{NY, 2008}	1st	1
{WA, 2008}	2nd	3

Figure 3.2: T_1's subspaces and its ranks.

Example 4 (CONTEXT) *We use the multidimensional model to accommodate typical decision support data. Figure 3.1 illustrates a sample fact table, where the four dimensions are categorized by two subspace dimensions (*Location, Year*), an object dimension (*Object*), and a score dimension (*Score*). Note that the subspace dimensions and the object dimension contain categorical values, and the score dimension is numerical. Assume that our target object for promotion is T_1. Figure 3.2 lists T_1's 6 subspaces and the corresponding rank of T_1 in each subspace. Here we assume that the rank of T_1 in each subspace is derived by ordering all objects in the subspace in descending order according to the* SUM *aggregate score. For instance, in $\{WA\}$ T_1 is ranked 3rd because we have $SUM(T_2) = 2.0 > SUM(T_3) = 1.3 > SUM(T_1) = 0.8$. Similarly, in $\{2008\}$ we have T_1's rank being 1st because $SUM(T_1) = 1.3 > SUM(T_2) = 1.0 > SUM(T_3) = 0.7$. Figure 3.2 also lists the number of distinct objects each subspace contains (i.e.,* ObjCount*). For example, $\{NY\}$ contains two objects, T_1 and T_3, since T_2 is not associated with any tuple matching the value "NY".*

In different contexts, we can distinguish between two types of object rankings. The first type of ranking is called *aggregate score-based ranking*. That is, the computation of an object's rank in any subspace involves dynamic score aggregation. This is illustrated in the above example, as T_1 is ranked lower than T_2 and T_3 in $\{WA\}$ but higher than them in {2008}. In principle, an object may or may not be ranked higher than another object in different subspaces because their scores are dynamically aggregated. The second type is called *constant score-based ranking*, where each object has a fixed global score. In other words, an object's score is constant across all subspaces. Therefore, objects' relative ranking will be fixed in any subspace (objects not appearing in a subspace will not be ranked). For example, in a product hierarchy where each product has a fixed weight, the relative ranking between the products will be fixed. In this study, we focus on

the former type, aggregate score-based ranking, which generalizes the latter one.

In order to quantify how well a subspace can serve the promotional purposes for the target object, we need to define the key notion of *promotiveness*. Intuitively, a subspace in which the target object is highly ranked would have high promotional value. However, this may not be the whole story.

Example 5 (PROMOTIVENESS) *Let us continue with our running example in Figure 3.2. Observe that, {2008} can be considered as a promotive subspace because the target object T_1 is ranked 1st in it, which is much better for promotion than the full space {*}, where T_1 is ranked the last. However, also observe that, although T_1 has equal ranks in {2008} and {NY}, these two subspaces may not be considered equally promotive, because in {2008} T_1 has two competitor objects but in {NY} there is only one. This intuitively indicates "more competition" in {2008} and thus it is often desirable that {2008} be considered more promotive than {NY}. For the same reason, even though T_1 is ranked 1st in {NY, 2008}, this subspace may not be considered promotive because T_1 has only one object.*

That the target object is highly ranked in a subspace does not necessarily suggest that the subspace is promotive, because other factors such as the number of competitors would affect its promotional value as well. Therefore, we need to support a class of measures to gauge the promotiveness of subspace. Instead of solely relying on rank, another element coined *subspace significance* is considered. The subspace significance is rank-independent and it models the interestingness of a subspace itself. The class of measures enables users to model context-specific semantics. However, a potential *spurious promotion* problem may arise when seemingly promotive subspaces are actually caused by random noises. To tackle the spurious promotion problem, statistical methods based on the analysis of variance are proposed as an integral part of our solution.

The promotion query problem presents the following computational challenges. First, a d-dimensional data set (*i.e.*, d subspace dimensions) has an exponential number of subspaces. A naive approach that enumerates and aggregates all subspaces would be prohibitively expensive. Second, the promotiveness measure is neither monotonic nor anti-monotonic. For example, in Figure 3.2, subspace {2008} may have a higher promotiveness measure value than both its child subspace {NY, 2008} and its parent subspace {*}. This means that establishing an early stopping condition is difficult: we cannot prune a subspace even when the target object has a low rank in it, because

the object may have a high rank in some children subspaces. Such non-monotonicity prevents us from utilizing existing aggregate computation methods, which require monotonicity of the measure. Third, the promotiveness measure is holistic [35], which means that computing the promotiveness measure requires us to aggregate *all* objects in any subspace to derive the target object's rank. Thus, the cost for computing the promotiveness measure is high. Also, the holistic property makes shared computation of the promotiveness measure across the subspace lattice difficult.

While answering promotion query is challenging, we develop algorithms that significantly outperform baseline solutions, making promotion analysis feasible for large-scale applications. We first propose a generic PromoRank framework, and then develop the following optimization techniques by exploiting the fact that users are only interested in the *top* subspaces where the target object has *top* ranks. (i) *Subspace pruning*: Users only desire the most promotive subspaces, so aggregations in many "unpromising" subspaces could be wasted. To avoid the cost of unnecessary aggregations, we establish upper and lower bounds for the promotiveness measure by utilizing the parent-child relationships among subspaces. The results of subspaces which have already been aggregated may be reused to prune unseen ones with little overhead. (ii) *Object pruning*: In many real data sets, object scores follow power-law distributions. When computing a target object's rank in a subspace, only objects with aggregate score larger than that of the target object would affect the target object's rank, whereas objects in the long tail would not. Therefore, they can be chopped off at an early stage, thereby reducing subsequent aggregation and ranking cost. (iii) *Promotion cube*: We develop a compact materialization strategy to further optimize the efficiency of query processing, which complements the online algorithms and seeks the middle ground between space overhead and query execution time. All these techniques are seamlessly integrated into the PromoRank framework. The contributions of this chapter are summarized as follows.

- Present the promotion analysis problem and its uses. To the best of our knowledge, this is the first work to systematically study the problem;
- The notion of subspace promotiveness is formalized (Section 3.2);
- Efficient query execution algorithms are proposed in the PromoRank framework (Section 3.3); a compact cube structure, called *promotion cube*, is proposed to further speedup query

processing (Section 3.4);

- Discuss statistical methods for preventing spuriously promotive results (Section 3.5);
- Verify the quality of promotion analysis using two real-world data sets. An extensive performance study shows that our proposed techniques are much more efficient than baseline algorithms (Section 3.6).

The remainder of this chapter is organized as follows. Section 3.2 formally introduces the promotiveness measure and the problem definition. Section 3.3 discusses the PromoRank framework with pruning techniques. Section 3.4 proposes the promotion cube technique. Section 3.5 discusses the method for removing spuriously promotive subspaces. Section 3.6 reports our experimental results. Finally, Section 3.7 concludes this chapter.

3.2 Problem Definition

3.2.1 Data Model

Consider a d-dimensional data set $\mathcal{D}$ consisting of a set of n base tuples, each having d (categorical) **subspace dimensions** $\mathcal{A} = \{A_1, A_2, \ldots, A_d\}$, an **object dimension** I_{obj}, and a **score dimension** I_{score}. Denote the object domain $dom(I_{obj})$ by $\mathcal{O}$, namely the complete set of object IDs (*e.g.*, in Example 3, $\mathcal{O} = \{T_1, T_2, T_3\}$). Let $dom(I_{score})$ be $\mathbb{R}^+$, the set of non-negative real numbers.

A **subspace** is defined as $S = \{a_1, a_2, \ldots, a_d\}$, where $a_i \in A_i$ or $a_i = *$ (star refers to the "any" value). S induces a projection of the data set $\mathcal{D}_S (\subseteq \mathcal{D})$ and a subspace of objects $\mathcal{O}_S (\subseteq \mathcal{O})$ (*e.g.*, in Example 3, $\mathcal{O}_{\{NY\}} = \{T_1, T_3\}$). A subspace $S_1 = \{a_1, a_2, \ldots, a_d\}$ is called a **child** subspace of $S_2 = \{b_1, b_2, \ldots, b_d\}$ iff there exists a j s.t. $a_j \neq * \wedge b_j = *$ and $a_i = b_i$ for any $i \neq j$. Conversely, S_2 is a **parent** subspace of S_1. For example, $\{NY\}$ is a parent subspace of $\{NY,\ 2008\}$, whereas $\{NY,\ 2008\}$ is child subspace of the former.

For this d-dimensional data, all subspaces can be partitioned into 2^d **cuboids** (or group-by's). We say that S belongs to a d'-dimensional cuboid $\mathcal{A}'$ denoted by $A'_1 A'_2 \cdots A'_{d'}$ iff S has non-star values in these d' dimensions and star values in the other $d - d'$ dimensions. These 2^d cuboids form a cuboid lattice, where in particular the apex cuboid denoted by "$*$" contains only the full

space $\{*, *, \ldots, *\}$. In our running example, there are four cuboids, "$*$", "*Location*", "*Year*", and "*Location/Year*", induced by the two subspace dimensions *Location* and *Year*. The cuboid "*Location*" contains two subspaces $\{WA\}$ and $\{NY\}$.

In subsequent discussions, assume a query **target object** $t_q \in \mathcal{O}$ is given by the user. Let $\mathbf{S}_q = \{S_q | t_q \in \mathcal{O}_{S_q}\}$ be the set of **target subspaces** where t_q occurs. These subspaces form a **target lattice**. In Example 3, we can see that the target object T_1 has 6 target subspaces as shown in Figure 3.2; subspace $\{2007\}$ is not a target subspace because T_1 does not occur in it.

3.2.2 A Unified Promotiveness Measure

For any target subspace S_q, we need to measure its promotional value for t_q. We assume that a higher rank of t_q in S_q should make S_q promotive. On the other hand, fixing t_q's rank, more competitors in S_q indicates that it has larger promotional value. To formalize the intuition, we define *promotiveness* as a class of composite measures.

Definition 2 (PROMOTIVENESS) *Given t_q and S_q, the promotiveness,* P*, can be defined as*

$$\mathsf{P}(t_q, S_q) = \mathsf{f}\big(\mathsf{Rank}(t_q, S_q)\big) \cdot \mathsf{g}\big(\mathsf{Sig}(S_q)\big), \tag{3.1}$$

where Rank *measures the rank of t_q in S_q based on a given monotone aggregate measure* $\mathbb{M}$*,* Sig *measures S_q's own* significance *(i.e., promotional value), and* f *and* g *are monotone normalization functions.*

P consists of three components: Rank, Sig, and f and g. We elaborate on each component and its assumptions in the following.

The rank measure

Definition 3 (RANK) *Given an aggregate measure* $\mathbb{M}$*, t_q's rank in S_q is defined as*

$$\mathsf{Rank} = \big|\{t | t \in \mathcal{O}_S \wedge t \neq t_q \wedge M^S(t) \rhd M^S(t_q)\}\big| + 1, \tag{3.2}$$

where $\rhd \in \{>, \geq, <, \leq\}$ and $M^S(t)$ denotes the aggregate score of object t in subspace S.

The rank is a deciding factor in promotiveness. We assume that $\mathbb{M}$ is a monotone measure such as *SUM*, *COUNT*, *MAX*, *etc.*. For simplicity, also assume $\triangleright$ = " > ", and our techniques can be extended to other relation operators.

The sig measure

The Sig measures the promotional value of the subspace itself, which is independent of what the target object t_q is. Some instances of this measure are TupleCount (the number of base tuples in the subspace), ObjCount (the number of distinct objects in the subspace), or Level (the number of star-values). For example, a larger ObjCount may suggest that the subspace is "more competitive", whereas a larger Level indicates that the subspace is more general. Here we assume Sig is monotone such that the Sig of a child subspace should be no greater than that of its parent subspace.

The normalization functions f and g

$\mathsf{f}(\cdot)$ and $\mathsf{g}(\cdot)$ are monotone normalization functions that combine Rank and Sig in order to derive a meaningful promotiveness value. Formally, "monotone" means that $\mathsf{r}_1 \leq \mathsf{r}_2 \Rightarrow \mathsf{f}(\mathsf{r}_1) \geq \mathsf{f}(\mathsf{r}_2)$ and $\mathsf{s}_1 \leq \mathsf{s}_2 \Rightarrow \mathsf{g}(\mathsf{s}_1) \leq \mathsf{g}(\mathsf{s}_2)$. That is, both higher rank and larger significance will lead to a larger promotiveness value.

Example instantiations

We illustrate several example instantiations of P to model certain semantics of the promotion query.

Enforcing iceberg constraint: One might use Rank to gauge the promotiveness of subspaces while enforcing a minimum support threshold *minsup* to filter out "small" subspaces. In this case, P could be like

$$\mathsf{P} = \mathsf{f}(\mathsf{Rank}) \cdot \mathcal{I}(\mathsf{TupleCount} \geq \mathit{minsup}),$$

where g is instantiated to the indicator function $\mathcal{I}$ (which returns 1 when the condition is true and 0 otherwise), and f can be any monotone function. Thus, any subspace not passing *minsup* will have 0 promotiveness. The iceberg constraint can avoid searching deep subspaces, which are often too sparse to be meaningful.

Percentile rank: Another useful instantiation of P is the percentile rank. For example, a subspace where t_q ranks to top-1% might be much more promotive than a subspace with percentile rank top-30%. The percentile rank is a common measurement in various applications, *e.g.*, studying test scores. Formally, we can instantiate P as

$$\mathsf{P} = \mathsf{Rank}^{-1} \cdot \mathsf{ObjCount},$$

such that a higher percentile rank of t_q would result in a larger promotiveness value. The percentile rank naturally captures the population size in each subspace so that smaller subspaces will be penalized.

Other functions: Users may propose other functions to customize P. For example, one may use *continuous* g function to penalize "small" subspaces, or assign static *weights* to subspaces (*e.g.*, larger weights are associated with the recent years than with the past).

3.2.3 The Promotion Query Problem

Now, the promotion query can be formulated as follows.

Definition 4 (Promotion Query) *Given a target object t_q, and a promotiveness measure P, return the top-k subspaces with the largest promotiveness values.*

Ties are broken arbitrarily. For clarity of presentation, in the subsequent discussions, we simplify the problem by letting $\mathbb{M}$ be *SUM* and P be $\mathsf{Rank}^{-1} \cdot \mathcal{I}(\mathsf{TupleCount} \geq minsup)$. These simplifications do not make our solutions less general to the query model. We discuss how to avoid spuriously promotive subspaces generated by random noise in Section 3.5.

3.3 The PromoRank Framework

In this section we discuss the PromoRank framework with two pruning methods: *subspace pruning* and *object pruning*. We start with a general framework that lays the foundation. This framework is based on the bottom-up computation method discussed in [8]. The general idea is as follows. It runs in memory and recursively partitions the data set according to some dimension, and objects

Algorithm 3.1: PromoRank(S, $\mathcal{D}$, $\mathcal{O}$, d_0)

Input:
Target object t_q, subspace S, and data set $\mathcal{D}$;
Object set in current subspace $\mathcal{O}$;
Previous partition dimension d_0.

Output:
Top-k promotive subspaces *Result*.

```
1  if |D| < minsup ∨ t_q ∉ O then return;
2  Compute Rank and P;
3  Update Results using (S, P);
4  for d' ← d_0 + 1 to d do
5      Sort D based on d'-th dimension;
6      foreach value v in d'-th dimension do
7          S' ← S ∪ {d' : v};
8          PromoRank(S', D_S', O_S', d');
9      end
10 end
```

Table 3.1: The PromoRank algorithm.

are aggregated in the subspace corresponding to each partition. Then the promotiveness measure value for that subspace is computed. Table 3.1 displays the outline of the framework, PromoRank, based on the example instantiation of promotiveness measure assumed in Section 3.2.3, *i.e.*, $\mathsf{P} = \mathsf{Rank}^{-1} \cdot \mathcal{I}(\mathsf{TupleCount} \geq minsup)$. The framework is divided into an aggregation phase and a partition phase.

Aggregation phase: In this phase, the Rank and P measures are computed for the input subspace S (Line 2). Then the target subspace S and its P value will be inserted into a priority queue which maintains the top-k results (Line 3). We now elaborate on the computation of P for S: Scan through the I_{obj} and I_{score} dimensions of the input base tuples in $\mathcal{D}$ and compute the aggregate score (*i.e.*, SUM) for each object in $\mathcal{O}$. This is implemented using a hash table keyed on object ID. The resulting hash table would have $|\mathcal{O}|$ entries which map each object to its aggregate score. Rank is computed directly by counting the number of aggregate scores strictly larger than t_q's aggregate score. Finally, P is set to Rank^{-1}.

Partition phase: The input data $\mathcal{D}$ is iteratively sorted according to the d'-th dimension in a depth-first search manner (Line 5). As a result, $\mathcal{D}$ can be projected into multiple partitions such that each partition corresponds to a distinct value on the d'-th dimension. A child subspace S' of S is defined on each partition (Line 7), and PromoRank recursively progresses over subspace S' and

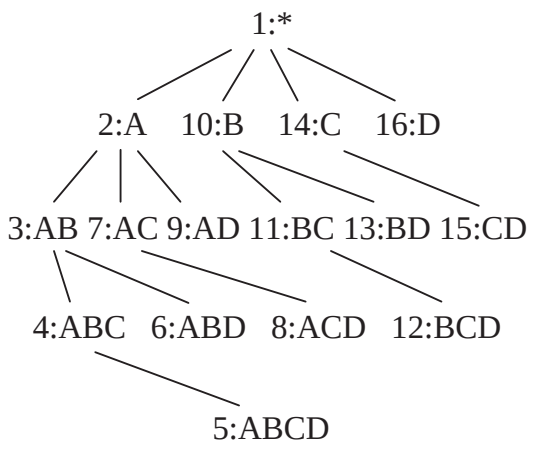


Figure 3.3: A cuboid tree with 4 dimensions and 16 cuboids. The aggregation order of the cuboids is labeled using the numbers.

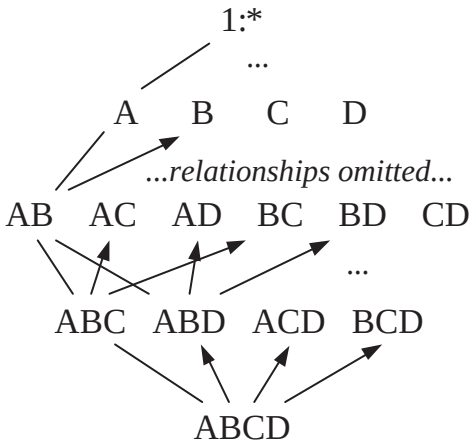


Figure 3.4: Exploiting interdependent subspace relationships to lower bound rank.

the corresponding data partition of S' (Line 8). Any subspace where the target object does not occur will be pruned (Line 1). Also, the iceberg constraint "TupleCount $\geq$ *minsup*" can be enforced as well (Line 1). Figure 3.3 displays an example recursive process for 4 dimensions at cuboid-level. For each cuboid, we label the order it is visited by PromoRank.

Although the PromoRank algorithm presented here supports an instance of the promotiveness measure with iceberg constraint, it is nevertheless generic that it is able to compute *any* promotiveness measure. Specifically, given any aggregate measure $\mathbb{M}$, Rank can be computed using the hash table in the same way as described above, and Sig can be directly derived from the input subspace S and/or the input data $\mathcal{D}$.

Analysis: At each recursion the aggregation runs in $O(|\mathcal{D}| + |\mathcal{O}|)$ time, due to the scan of the input data and hashing on objects. The partition phase runs in $O(|\mathcal{D}|)$ time because the input data was sorted during the last recursion. Overall there will be $|\mathbf{S}_q|$ recursions, so the total cost of the algorithm can be written as $C^{all} = \sum_{i=1}^{|\mathbf{S}_q|}(C_i^{par} + C_i^{agg})$, where C_i^{par} and C_i^{agg} are the partition and aggregation costs at the i-th recursion, respectively.

Notice that this algorithm computes *all* subspaces, and thus the overall cost C^{all} could be quite prohibitive for large data sets. Because users often ask for only the *top* subspaces where the target object has *top* ranks, we further develop optimization techniques in the subsequent sections.

3.3.1 Subspace Pruning

Now we discuss the subspace pruning technique. The key motivations for this technique are that (i) users are often only interested in top-k promotive subspaces, and (ii) the aggregate measure $\mathbb{M}$ is monotone (*e.g.*, *SUM*). Intuitively, it could be wasteful to perform aggregation for all subspaces. To prune out "unpromising" subspaces, the problem becomes to establish an upper bound for the promotiveness measure.

Given the set of target subspaces $\mathbf{S}_q$, the PromoRank algorithm iterates through each subspace in it in a sequential order, $S_1, S_2, \ldots, S_{|\mathbf{S}_q|}$ (illustrated in Figure 3.3), if both the dimension ordering and the value ordering on each dimension are fixed. At any time of the algorithm, the sequence of subspaces can be conceptually split into a list of seen subspaces which have already been aggregated, and a list of unseen subspaces which have not yet been aggregated. From the list of seen subspaces, we compute the current k-th largest promotiveness value as a threshold. For the list of unseen subspaces, we derive an upper bound promotiveness value for each of them using already aggregated results. Thus, any unseen subspace whose upper bound is less than the threshold can be pruned. To derive the upper bound, we utilize parent-child relationships between seen and unseen subspaces in the target lattice.

We first introduce some notation. At any time k of the algorithm, let

- $\mathbf{S}_{seen}$ denote the *list of seen subspaces* $\{S_1, \ldots, S_k\}$ $(0 \leq k \leq |\mathbf{S}_q|)$ which have been aggregated or pruned already;
- $\mathbf{S}_{unseen}$ denote the *unseen subspace list* $\{S_{k+1}, \ldots, S_{|\mathbf{S}_q|}\}$;
- $M^i(t)$ denote the *aggregate score* of object t $(\in \mathcal{O}_{S_i})$ in any subspace S_i $(1 \leq i \leq |\mathbf{S}_q|)$; in particular, $M^i(t_q)$ denote the aggregate score of the target object in S_i;
- $S_i.\mathsf{P}$, $S_i.\mathsf{Sig}$, and $S_i.\mathsf{Rank}$ denote the *exact measure values* of S_i $(1 \leq i \leq |\mathbf{S}_q|)$;
- $S_i.\overline{\mathsf{P}}$, $S_i.\overline{\mathsf{Sig}}$, and $S_i.\underline{\mathsf{Rank}}$ denote the *upper bound* P, *upper bound* Sig, and *lower bound* Rank of an unseen subspace S_i $(k+1 \leq i \leq |\mathbf{S}_q|)$;
- $\underline{\mathsf{P}}$ denote the *k-th largest promotiveness measure value of all seen subspaces.*

Now we consider the problem of computing a general upper bound for promotiveness, *i.e.*, computing $S_i.\overline{\mathsf{P}}$ given an unseen subspace S_i. By definition, $S_i.\overline{\mathsf{P}} = \mathsf{f}(S_i.\underline{\mathsf{Rank}}) \cdot \mathsf{g}(S_i.\overline{\mathsf{Sig}})$ because of the monotonicity of f and g. In general, $S_i.\overline{\mathsf{Sig}}$ can be computed based on S_i's seen parents since Sig is monotone (note that the iceberg constraint assumed is a specific instance of $\mathsf{g}(\mathsf{Sig})$). So now the problem is reduced to computing $S_i.\underline{\mathsf{Rank}}$. Because Rank is neither monotone nor convex, we cannot compute $S_i.\underline{\mathsf{Rank}}$ directly from S_i's parent subspaces. A trivial lower bound is 1, which assumes the best possible rank; however, this would not be able to provide any pruning power. Our idea here is to exploit the monotonicity of the aggregate measure in the subspace lattice.

Claim 1 (Rank Measure Lower Bound) *Let S_c be any child subspace of S, we can obtain a lower bound* Rank *measure as* $S.\mathsf{Rank} \geq |\{t | t \in \mathcal{O}_{S_c} \wedge M^{S_c}(t) > M^S(t_q)\}| + 1$.

The claim is clear as aggregate scores must be monotone across parent-child subspaces. Therefore, given an unseen subspace S_i and its seen child subspace S_j that has already been aggregated, we can compute the lower bound Rank for S_i as $S_i.\underline{\mathsf{Rank}} = \left|\{t | t \in \mathcal{O}_{S_j} \wedge M^j(t) > M^i(t_q)\}\right| + 1$, by using $M^j(t)$, namely the already aggregated object scores in the child subspace S_j, and $M^i(t_q)$, namely the target object's aggregate score in S_i. When there are multiple unseen child subspaces of S_i, there could be multiple lower bounds. In such cases, the maximum value is taken to provide the tightest bound. This way, the upper bound for promotiveness can be established, enabling our subspace pruning strategy.

PromoRank+, the subspace pruning algorithm based on the basic framework, is displayed in Table 3.2. However, as shown in Table 3.2, it is "flattened" for simplicity (*i.e.*, instead of showing a recursive algorithm, we use a for-loop to represent it). Note that in order to compute $S_i.\underline{\mathsf{Rank}}$ for any S_i, we need to first compute the target object's aggregate score in each target subspace (Line 2). Also, all bounds are initialized (Lines 3 and 5), and the base tuples pertaining to the target object are removed from the data (Line 6). Here computing $M^i(t_q)$ for all target subspaces is very efficient because the number of base tuples related to t_q often makes only a small portion of the data. This could be even more efficient if a clustered index has been built on the object dimension.

During partitioning, PromoRank+ iteratively aggregates each subspace (Lines 7–17). At each iteration, one of the following two cases happens. If the upper bound promotiveness of the current subspace is less than the k-th largest promotiveness value seen so far, the subspace can be pruned.

Algorithm 3.2: PromoRank+

```
1   for i ← 1 to |S_q| do /* initialization */
2       Compute M^i(t_q);
3       S_i.P̄ = S_i.Siḡ ← ∞, S_i.Rank ← 1;
4   end
5   P ← 0;
6   D = D\{t_q};
7   for i ← 1 to |S_q| do
8       if S_i.P̄ > P then /* if not pruned */
9           Compute S_i.P, update P and top-k results;
10          foreach S_i's descendent S_j (j > i) do
11              S_j.Siḡ ← S_i.Sig;
12              Update S_j.P̄;
13          foreach S_i's parent S_j (j > i) do
14              Update S_j.Rank and S_j.P̄;
15      end
16      Sort and partition; /* recursive */
17  end
```

Table 3.2: The subspace pruning algorithm.

In this case, its aggregation step can be avoided (*i.e.*, skips Line 9–14). Otherwise, object score aggregation must be performed to obtain the exact P measure value (Line 9). After that, S_i.Sig can be used to upper bound the Sig measure of all S_i's descendent subspaces (Lines 10–11). Meanwhile, the resulting aggregate scores can be reused to derive Rank for its unseen parent subspaces (Lines 13–14). The updated bounds of Sig and Rank of unseen subspaces are then propagated to the upper bound of P (Lines 12 and 14).

Example 6 *Figure 3.4 displays the computation of lower bound* Rank *in a lattice view. Subspaces are recursively aggregated in the order* $A \rightarrow \cdots \rightarrow ABCD \rightarrow ABD \rightarrow \cdots$. *When, for example, a subspace in ABC has just been aggregated, the* Rank *of its corresponding unseen parents in AC and BC can be updated; we do not update AB because it is seen. Similarly, aggregated results in* ABD *can be reused for AD and BD.*

Subspace	Objects ($\mathcal{O}_S$) and their aggregate scores ($M^S(t)$)	$M^S(t_q)$	Rank	P
$S_1 = \{a\}$	$t_6(1.2)$ $t_3(1.0)$ $t_1(0.7)$ $t_7(0.7)$ $t_4(0.3)$ $t_5(0.3)$ $t_2(0.2)$	0.7	3	1/3
$S_2 = \{ab\}$	$t_6(0.7)$ $t_3(0.6)$ $t_7(0.6)$ $t_1(0.4)$ $t_4(0.3)$ $t_2(0.2)$ $t_5(0.2)$	0.6	2	1/2
$S_3 = \{abc\}$	$t_3(0.6)$ $t_6(0.5)$ $t_7(0.3)$ $t_1(0.1)$ $t_5(0.1)$	0.3	3	1/3
$S_4 = \{ac\}$	$t_3(0.8)$ $t_6(0.7)$ $t_1(0.6)$ $t_7(0.4)$ $t_5(0.2)$ $t_2(0.1)$ $t_4(0.1)$	0.4	4	1/4

Figure 3.5: Example subspaces and their aggregated results.

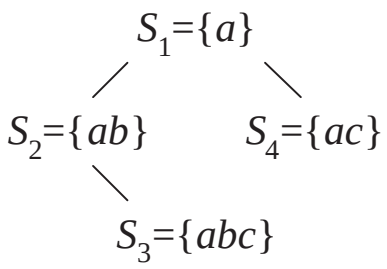


Figure 3.6: A subtree of subspaces rooted at S_1.

Figures 3.5 and 3.6 further show a concrete example. Assume the target object is t_7. *Also assume* $\mathsf{P} = \mathsf{Rank}^{-1}$. *When* PromoRank *is executed, 4 target subspaces will be aggregated in the order* $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4$. *The aggregate scores in each subspace are listed in Figure 3.5, from which* Rank *and* P *can be derived.*

Now suppose PromoRank+ *is executed on the same data and* $S_1 \rightarrow S_2 \rightarrow S_3$ *have just been aggregated. Because* S_3 *has an unseen parent subspace* S_4, *we can reuse* S_3*'s result to compute* $S_4.\underline{\mathsf{Rank}} = \left|\{t|t \in \mathcal{O}_{S_3} \wedge M^3(t) > M^4(t_q)\}\right| + 1$. *Since we have* $M^4(t_q) = 0.4$ *during initialization,* $S_4.\underline{\mathsf{Rank}} = |\{t_3, t_6\}| + 1 = 3$. *In this case, if we want to find the top-1 promotive subspace, we would have* $\underline{\mathsf{P}} = S_2.\mathsf{P} = 1/2$ *(i.e., 1/2 being the largest* P *seen so far), and we can safely prune out* S_4 *because* $S_4.\overline{\mathsf{P}} = 1/4 < \underline{\mathsf{P}}$.

Analysis: $S_i.\underline{\mathsf{Rank}}$ can provide effective pruning power when $M^i(t_q)$ is relatively small, since in such cases $S_i.\underline{\mathsf{Rank}}$ would be bounded away from top ranks. Because our goal is to find the top-k most promotive subspaces, many target subspaces with small $M^i(t_q)$ can be pruned, leading to lower aggregation cost. Another advantage of subspace pruning is that subspaces are pruned with little overhead: First, computing t_q's aggregate scores beforehand incurs no redundant cost because all base tuples related to t_q are removed subsequently. Second, no intermediate aggregate results need to be stored because the lower bound Rank is computed during aggregation.

3.3.2 Object Pruning

In the basic PromoRank framework, the Rank measure is computed in a holistic manner. That is, for each subspace, the complete set of objects in that subspace are aggregated and compared in order to derive Rank for the target object. However, this often incurs huge waste as only the objects

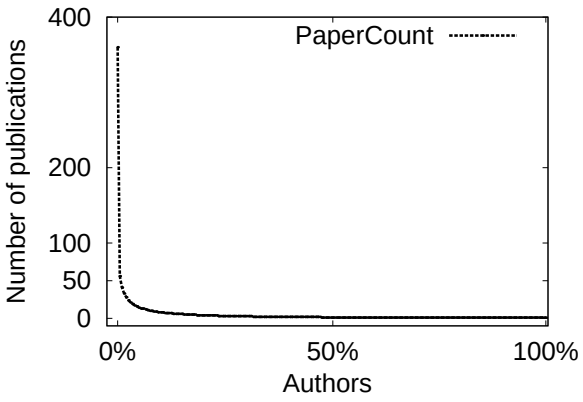


Figure 3.7: The power-law distribution of aggregate scores in DBLP. All authors are ordered by the number of publications.

which have larger aggregate scores than that of the target object would determine Rank, regardless of how many objects there are. We motivate this pruning technique using a typical real-world example: Figure 3.7 displays a power-law distribution in the DBLP data set [1], where the X-axis (object) corresponds to more than $450K$ authors and the Y-axis (aggregate score) represents the number of publications each author has. All authors are ordered descendingly by their aggregate score. We observe that most authors in the long tail would not affect the computation of Rank of the target object, and thus performing aggregation at each recursion for *all* objects and recursively passing these objects to the next child subspace would be unnecessary. This motivates us to develop the object pruning technique, which aims to determine the *minimal set* of objects that affect the computation of Rank, so that all remaining objects that do not fall into this set are pruned early.

For a target subspace S_i, let $S_i.MinScore$ denote the minimum aggregate score of the target object in the S_i's subtree. Here the *subtree* is defined as the depth-first search tree induced by the recursive process, as illustrated in Figure 3.3. For example, in Figures 3.6 and 3.5, S_1's subtree consists of S_1 through S_4, and $S_1.MinScore = \min\{0.7, 0.6, 0.3, 0.4\} = 0.3$.

Therefore, given a subspace S_i and any object $t \in \mathcal{O}_{S_i}$, t can be pruned if $M^i(t) \leq S_i.MinScore$. This is because, for any subspace S_j in S_i's subtree, $M^j(t) \leq M^i(t) \leq S_i.MinScore \leq M^j(t_q)$, which means that t's aggregate score is no greater than the target object's aggregate score in S_j. In other words, t would not affect S_j.Rank, and this holds true for any S_j in S_i's subtree during the depth-first search process. Therefore, t can be pruned from $\mathcal{O}_{S_i}$. Once t is pruned, its corresponding base tuples in the current data partition $\mathcal{D}_{S_i}$ can be pruned as well.

Algorithm 3.3: PromoRank++

```
1   for i ← 1 to |S_q| do /* initialization */
2         S_i.P̄ = S_i.Siḡ ← ∞;
3         S_i.Rank ← 1;
4         Compute M^i(t_q);
5   end
6   Compute S_i.MinScore for each subspace S_i;
7   Set the pruning threshold P ← 0;
8   D ← D\{t_q};
Procedure: PromoRank++(S, D, O, d_0)
9   if S.P̄ > P then /* subspace pruning */
10        Compute S.Rank, S.Sig, and S.P;
11        Update P and top-k results;
12        foreach S' descendent S_j do
13              S_j.Siḡ ← S_i.Sig;
14              Update S_j.P̄;
15        end
16        foreach S' parent S_k (k > i) do
17              Update S_k.Rank and S_k.P̄;
18        end
19        Compute O_min using S.MinScore;
20        D ← D\O_min; /* object pruning */
21  end
22  /* partition and aggregate*/
23  for d' ← d_0 + 1 to d do
24        Sort D based on d'-th dimension;
25        foreach child target subspace S' along d'-th dimension do
26              PromoRank++(S', D_S', O_S'\O_min, d');
27        end
28  end
```

Table 3.3: The complete query execution algorithm.

Example 7 *Continue with the example in Figures 3.5 and 3.6, where* S_1*.MinScore*=0.3*. Suppose the aggregate scores of objects in* $\mathcal{O}_{S_1}$ *have just been computed for* S_1*. By checking these scores, one can see that* t_4 *(0.3),* t_5 *(0.3), and* t_2 *(0.2) have scores no greater than* S_1*.MinScore. This means that* t_4*,* t_5*, and* t_2 *would not affect the computation of the rank measure in any subspace in* S_1*'s subtree (i.e.,* t_4*,* t_5*, and* t_2 *would not affect the target object's rank in* S_1*,* S_2*,* S_3*, and* S_4*). Therefore, these three objects can be safely pruned so that they do not need to be aggregated in the unseen subspaces* S_2*,* S_3*, and* S_4 *in the subtree. In Figure 3.5, we can see that the* Rank *of the target object* t_7 *in each of the four subspaces are not decided by* t_4*,* t_5*, or* t_2*.*

This object pruning strategy can be pushed down into the PromoRank algorithm recursively and be integrated with subspace pruning. The complete query execution algorithm, PromoRank++, is shown in Table 3.3. We highlight its differences from PromoRank+ as follows. (i) Initializing *MinScore*: This happens at the initialization stage (Line 6), which can be efficiently computed bottom-up in the cuboid tree. (ii) Recursive object pruning: At the end of each aggregation phase (except for those subspaces without any child subspace), identify the set of objects to be pruned $\mathcal{O}_{min} = \{t | t \in \mathcal{O}_S \wedge M^S(t) \leq S.MinScore\}$ (Line 19). Then, prune all base tuples related to $\mathcal{O}_{min}$ from the current data partition (Line 20).

Analysis: By pruning objects and data, the cost of both aggregation and partitioning would be greatly reduced. This pruning strategy introduces little overhead as $\mathcal{O}_{min}$ can be directly computed from aggregated results, and the additional space overhead is $O(1)$ per subspace. However, using this object pruning strategy, objects and data are pruned at early stage and we may not be able to compute some Sig measures like ObjCount exactly (other measures like Level remain unaffected). Two techniques may be used to address this problem. First, approximate Sig using selectivity estimation techniques like [79]. Because promotiveness is not sensitive to Sig, such approximation often would not harm result quality. In our experiments we verified that a simple approximation method could work very well. Second, one may materialize Sig of subspaces which pass a *minsig* threshold so that Sig can be accurately obtained.

3.4 Promotion Cube

In practice, interactive and explorative analysis requires very short response time of queries. To further speedup promotion query processing for real-world applications, we propose a *Promotion Cube* technique to complement the online algorithms. The goal of the promotion cube is to (i) quickly locate promotive subspaces, and (ii) effectively prune out less promotive subspaces for an arbitrary target object. Toward this end, we exploit two types of correlations. First, the target object's rank is strongly correlated with its promotiveness value. In other words, high rank likely leads to large promotiveness. Second, a promotive subspace is unlikely to be insignificant. Therefore, the promotion cube precomputes only subspaces with Sig above a certain threshold *minsig*, which is similar to an iceberg cube. In each precomputed subspace, instead of directly materializing

objects and their promotiveness values, we materialize a set of order statistics. Specifically, only a very small set of the largest aggregate scores is precomputed without considering actual object IDs. A key advantage of this structure is that the threshold *minsig* and the size of the order statistics do not limit the capability of query processing. Any top-k promotive subspace can be discovered for any target object even if the subspace does not satisfy the threshold or if the target object's aggregate score in that subspace is not precomputed.

The definition of the promotion cube structure is as follows. Consider data set $\mathcal{D}$ and aggregate measure $\mathbb{M}$. Assume two cube parameters, *maximum rank* K and *minimum significance threshold* *minsig*, are given. Another optional parameter, *cell size* K', will be discussed shortly.

Definition 5 (Promotion Cell) *Given a subspace S, a promotion cell $S.PCell = (M_i)_{i=1}^{K}$ is defined as the sequence of the top-K largest object aggregate scores in S.*

Definition 6 (Promotion Cube) *The promotion cube $\mathbb{D}$ consists of a set of triples in the format $(S, PCell, \mathsf{Sig})$, where any subspace S must pass the* minsig *threshold. Formally, $\mathbb{D} = \{S : (S.PCell, S.\mathsf{Sig}) | S.\mathsf{Sig} \geq minsig\}$.*

If $K = |\mathcal{O}|$ and $minsig = 0$, $\mathbb{D}$ becomes equivalent to a full cube because all subspaces and all object scores would be materialized. Obviously, the storage cost would be very expensive because there could be an exponential number of subspaces as well as a large number of objects. In practice, we select $minsig > 0$, so that many subspaces being insignificant will be ignored, resulting in a small number of precomputed subspaces. Further, we have $K \ll |\mathcal{O}|$, meaning that the size of each promotion cell is also small. As a result, the promotion cube would be much more compact than the corresponding iceberg cube, which is in turn much smaller than the corresponding full cube. In fact, even for large data sets, the promotion cube is able to reside in main memory. Also, it can be efficiently computed using existing techniques [37].

The promotion cube contributes to the online query execution by enhancing subspace pruning. It provides non-trivial upper bound and/or lower bound promotiveness scores to precomputed subspaces. More specifically, given target object t_q and target subspace S, the computation of the upper bound promotiveness $S.\overline{\mathsf{P}}$ (and the lower bound $S.\underline{\mathsf{P}}$) can be separated into the following 3 cases.

- $S \in \mathbb{D} \wedge M^S(t_q) \in S.PCell$: We can obtain the exact values of $S.\mathsf{Sig}$, $S.\mathsf{Rank}$, and $S.\mathsf{P}$ (*i.e.*, $S.\overline{\mathsf{P}} = S.\underline{\mathsf{P}} = S.\mathsf{P}$);

- $S \in \mathbb{D} \wedge M^S(t_q) \notin S.PCell$: Here exact $S.\mathsf{Sig}$ can be obtained, and Rank can be lower-bounded as $S.\underline{\mathsf{Rank}} = K + 1$. $S.\overline{\mathsf{P}}$ can be computed correspondingly;

- $S \notin \mathbb{D}$: This means that Sig can be upper-bounded as $S.\overline{\mathsf{Sig}} = minsig$. $S.\overline{\mathsf{P}}$ can be computed accordingly.

The integration of the promotion cube with the query execution algorithm PromoRank++ (Table 3.3) is as follows. First, when initializing the promotiveness bound for each target subspace S_i (Line 2), instead of assigning a trivial value, the upper bound $S_i.\overline{\mathsf{P}}$ and in some cases the lower bound $S_i.\underline{\mathsf{P}}$ can be computed as described above. Second, instead of initializing $\underline{\mathsf{P}}$ to be 0 (Line 6), we let it be the k-th largest lower bound promotiveness among all target subspaces (*i.e.*, the k-th largest $S_i.\underline{\mathsf{P}}$). The rest of the algorithm remains unchanged.

While in principle each promotion cell in the promotion cube contains only the K largest aggregate scores, $M_1, M_2, \ldots, M_K$, a heuristic to further reduce space is to materialize only a subset of these scores. Given another cube parameter, cell size K', where $K' \leq K$, one can select k' aggregate scores out of the K scores at evenly spaced ranks. For example, if $K = 50$ and $K' = 5$, we materialize $M_{10}, M_{20}, \ldots, M_{50}$. For a target object, its upper and lower bound ranks are obtained by comparing its aggregate score with the K' materialized scores, and then the promotiveness bounds can be computed.

Unlike traditional data cubes which directly store aggregated results that users are interested in, the promotion cube complements the online query execution through offline preprocessing. Since the promotion cube will sit in memory, the additional query execution overhead for computing the bounds for target subspaces can be neglected. The parameters, *minsig*, K, and K', allow users to control the tradeoff between the online and offline costs so that users may select these parameters to yield the desired tradeoff. On the other hand, the parameters do not restrict the freedom of queries; that is, given $\mathbb{M}$, the promotion cube supports arbitrary promotion queries and guarantees the correctness of results.

3.5 Avoiding Spurious Promotion

In our query model, the promotiveness measure P consists of two components, the Rank measure, to make the target object prominent in a result subspace, and the Sig measure, to penalize subspaces being too sparse or too specific. There are cases, however, that the promotiveness measure fails to guarantee the meaningfulness of results. Let us consider the following motivating example.

Example 8 (Spuriously Promotive Subspaces) *Michael Jordan is the top scorer among all players born in February and the top scorer on sunny days.*

The example illustrates that the two subspaces $\{BirthMonth = February\}$ and $\{Weather = Sunny\}$, both having high target object ranks and are neither too sparse nor too specific, are nevertheless meaningless. Clearly, there exists no causal relationship between *BirthMonth/Weather* and NBA players' ranking. Such kind of "promotive" subspaces cannot be justified and thereby having no true promotional value. We call them *spuriously promotive subspaces* and formally discuss how to avoid them in this section.

We observe that the key difference between a spuriously promotive subspace and a truly promotive one lies in that the former involves at least one *spurious dimension*. For example, *Weather* is a spurious dimension when promoting the player. Intuitively, such a spurious dimension has no correlation with object ranking. When conditioning on spurious dimension values, the object score distribution would not be significantly changed; in other words, the promotion of the target object would be merely due to random perturbations of ranking.

Definition 7 (Spurious Promotion) *A subspace dimension is* spurious *when it is statistically independent of the score distribution. Any promotive subspace which contains a non-star value in some spurious dimension is considered to be* spuriously promotive.

Now, given a data set $\mathcal{D}$ (n base tuples), a subspace dimension A, and the score dimension I_{score}, our goal becomes to determine whether or not A and I_{score} have any correlation. Suppose A induces a partitioning of all base scores in I_{score} into θ (*i.e.*, A's cardinality) groups of scores, denoted as $\{s_j^1\}_{j=1}^{\varphi_1}$, $\{s_j^2\}_{j=1}^{\varphi_2}$, ...,$\{s_j^\theta\}_{j=1}^{\varphi_\theta}$, where φ_i denotes the cardinality of the i-th group ($1 \leq i \leq \theta$) and $\sum_i \varphi_i = n$. These groups are considered as samples drawn from θ underlying populations. Note

that a special case is that when A induces a partitioning of all objects, we can instead partition their aggregate scores into θ groups and hence $\sum_i \varphi_i = |\mathcal{O}|$. We employ the Analysis of Variance (ANOVA) test [51] to determine if significant difference exists between these group means or they only differ by chance. The idea of the ANOVA test is to compare these sample scores' between-group sum of squared deviation (SS_B) to their within-group sum of squared deviation (SS_W). The closer they are, the higher the probability that dimension A has no effect on I_{score}. Specifically, let the null hypothesis H_0 state that the mean is the same for all groups. Let σ_i and μ_i be the sum and average score of the i-th group, respectively. Then let

$$SS_B = \sum_i \frac{\sigma_i^2}{\varphi_i} - \frac{(\sum_i \sigma_i)^2}{n},$$
$$SS_W = \sum_i \sum_j (s_j^i - \mu_i)^2.$$

The F-ratio for dimension A can be calculated:

$$F(A) = \frac{SS_B/(\theta - 1)}{SS_W/(n - \theta)}.$$

Let $F_c(A)$ be the sample scores' corresponding critical value determined by $\theta - 1$, $n - \theta$ (or $|\mathcal{O}| - \theta$ for the special case), and a given Type I error probability α (*e.g.*, 0.05). If $F(A) \geq F_c(A)$, H_0 can be rejected; otherwise, we conclude that A does not significantly influence I_{score} and thus A is a spurious attribute.

ANOVA assumes normality of score distribution and homogeneity of score variances across different groups. When the assumptions are violated, one may alternatively apply power transformations to the data or employ non-parametric methods like Kruskal-Wallis test [51].

Computational complexity: To avoid spurious promotion, we preprocess the given data set $\mathcal{D}$ by removing all spurious dimensions based on the ANOVA method and, during query execution, avoid using any spurious dimension.

This preprocessing entails one pass over the data to compute the F-ratio for each subspace dimension. This requires $O(nd)$ time and $O(\sum_{i=1}^{d} \theta_i)$ space complexity, where d is the total number

of subspace dimensions and θ_i is the cardinality of the i-th subspace dimension.

3.6 Experimental Evaluation

In this section, our conduct comprehensive performance evaluation and our goal is to (i) verify the effectiveness of the promotion query through case study, and (ii) analyze the performance of our proposed algorithms in terms of both query execution time and storage space used. We break down our report of the evaluation results into 3 data sets and summarize the results as follows:

NBA data set [2] (Section 3.6.2) Conduct case study to show how NBA players can be effectively promoted. We also show the results on avoiding spurious promotion. Because this data set is small, no performance result will be reported.

DBLP data set [1] (Section 3.6.3) Conduct both case and performance studies. We confirm that the search results match our intuition well, and the proposed algorithms perform significantly better than baseline algorithms.

TPC-H [3] (Section 3.6.4) By generating synthetic data sets using a wide range of parameters, we show that the proposed algorithms consistently outperform the baseline ones.

3.6.1 Implementation

All experiments were done on a machine with a Pentium 3GHz processor, 2GB of memory, and 160G hard disk. We implemented the PromoRank, PromoRank++, and PromoCube algorithms. Our performance evaluation is based on two measures: *query execution time* and *space overhead*. PromoRank is considered a baseline for query execution time, while to evaluate the space overhead of PromoCube, we compare it to a traditional iceberg cube which, for each subspace passing some given *minsig* threshold, fully materializes all object aggregate scores.

The source code was written in C# and compiled using Microsoft Visual C# 2008 in Windows XP. All query processing algorithms were executed in the main memory without any disk access (PromoCube resides in memory).

Target object	Top-3 promotive subspaces	Rank	ObjCount	Top-%
Michael Jordan	{*}	3	3460	0.09%
	{Position=Guard}	1	1417	0.07%
	{Team=Chicago Bulls}	1	283	0.35%
	{Year=1984 (ties: 1986-1992, 1995-1997)}	1	380	0.26%
LeBron James	{*}	251	3460	7.3%
	{CareerStage=Young, Position=Guard}	4	1385	0.3%
	{CareerStage=Young}	14	3387	0.4%
	{Team=Cleveland Cavaliers}	1	278	0.4%
Al Jefferson	{*}	827	3460	23.9%
	{Year=2007}	11	451	2.4%
	{Position=Forward, Team=Boston Celtics}	24	139	17.3%
	{Team=Minnesota Timberwolves}	27	143	18.9%
Raymond Felton	{*}	930	3460	26.9%
	{Year=2006, CareerStage=Young}	5	142	3.5%
	{Year=2005, CareerStage=Young}	9	139	6.5%
	{Coach=Bernie Bickerstaff}	10	128	7.8%
Carlos Delfino	{*}	1337	3460	38.6%
	{Position=Guard, Team=Detroit Pistons}	33	132	25.0%
	{Coach=Flip Saunders}	32	92	34.8%
	{Team=Toronto Raptors}	36	132	27.3%

Table 3.4: A case study on the NBA data.

3.6.2 The NBA Data Set

Data characteristics: The data set was downloaded from [2]. It contains 18050 base tuples, each recording a player's statistics in a particular year. Each base tuple contains 5 subspace dimensions, namely *Year* (62 values), *CareerStage* (2 values, "*Young*" or otherwise), *Position* (3), *Team* (68), and *Coach* (220). We used *PlayerID* (3460) as the object dimension, and used statistics dimensions such as *Points*, *Rebounds* as score dimensions.

Promotiveness measure: *SUM* was chosen as the measure $\mathbb{M}$ to aggregate over *Points* in order to generate ranking. The promotiveness measure was set to $\mathsf{P}_1 = -\mathsf{Rank} - 2^{-\lceil \log(\mathsf{TupleCount}/n) \rceil}$, where $2^{-\lceil \log(\mathsf{TupleCount}/n) \rceil}$ is a penalty equivalent to 2^l (l being a non-negative integer) when the subspace selectivity $\frac{\mathsf{TupleCount}}{n}$ falls in range $(\frac{1}{2^{l+1}}, \frac{1}{2^l}]$.

We first conduct a case study. Table 3.4 shows the top-3 promotive subspaces for 5 representative players using the promotiveness measure discussed above. For comparison, it shows their global and local ranks, as well as their precise ranks in percentage. We find these results to match the reality very well. For example, the case for Michael Jordan has been used in the Example 2

in Section 3.1. Note that, for Michael Jordan, there are 11 subspaces which are ranked third since they have the same promotiveness value. For LeBron James, he is ranked only 251st among all players (the percentile rank is 7.3%); however, the promotion analysis results reveal more exciting facts such as that James being a talented young player (*i.e.*, 14th out of 3387 young players). For other globally low-ranked players listed in the table, we can see that their promotive subspaces make sense as well.

The above results are obtained using PromoRank++, where the object pruning method estimates the TupleCount of a subspace using the product of the selectivities of the subspace' dimension values. In fact, the results in Table 3.4 are accurate, because the promotiveness measure being used is not sensitive to selectivity.

Handling spurious promotion: Figure 3.8 shows the effectiveness of the ANOVA test for detecting spurious dimensions. In addition to the 5 subspace dimensions mentioned earlier, another 2 dimensions *BirthMonth* (12) and *RandomDim* (100) were also considered here for comparison. *BirthMonth* records a player's birth month and was extracted from the original data set [2]. *RandomDim* is a dimension that was attached to the original fact table using a uniform random number generator. The score dimension was set to *Rebounds*. Figure 3.8 displays the F-ratio for each of the 7 dimensions (on log scale/in a decreasing order). The corresponding critical values at $\alpha = 0.05$ for the 7 dimensions are shown in the figure for comparison. In the figure, we can see that the null hypothesis for *BirthMonth* and *RandomDim* cannot be rejected since their F-ratios are smaller

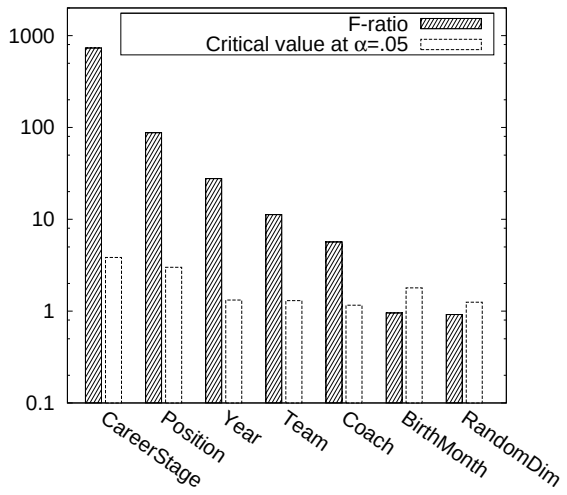


Figure 3.8: Correlation test on NBA dimensions.

Target object	Top-3 subspaces by P_1	Rank	ObjCount	Top-%
David DeWitt	{*}	376	451316	0.08%
	{Database}	16	65321	0.02%
	{1990}	2	13170	0.02%
	{SIGMOD}	2	3519	0.06%
Yufei Tao	{*}	3325	451316	0.74%
	{Database, 2003}	11	6707	0.16%
	{Database, 2004}	18	8877	0.20%
	{ICDE}	30	4822	0.62%
Target object	Top-3 subspaces by P_2	Rank	ObjCount	Top-%
David DeWitt	{PDIS}	1	318	0.31%
	{Database, SIGMOD}	1	1784	0.06%
	{Database, 1985}	1	556	0.18%
Yufei Tao	{ICDE, 2004}	1	334	0.30%
	{Data Mining, Info. Retrieval, 2004}	2	690	0.29%
	{SIGMOD, 2008}	5	471	1.06%

Table 3.5: Promotion query results on the DBLP data using different promotiveness measures.

than the corresponding critical values. This means that these two dimensions have no significant correlation with the score dimension and thus the subspace object ranking. On the other hand, the remaining 5 subspace dimensions have F-ratios significantly larger than the critical values, exhibiting that they are strongly correlated with player rankings. These results have empirically verified that spurious dimensions indeed can be separated from meaningful subspace dimensions by the statistical test.

3.6.3 The DBLP Data Set

Now let us report the results on the DBLP data set [1].

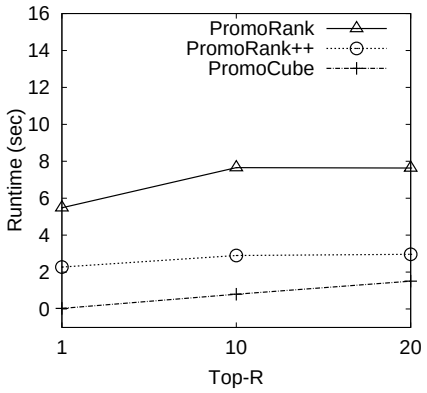
Data characteristics: We extracted 1,763,888 base tuples from the DBLP data set, each in the format (*Author, Conference, Year, Title*). The cardinalities of *Author*, *Conference*, and *Year* are 451316, 2506, and 50, respectively. We used *Author* as object dimension and consider *Conference* and *Year* as subspace dimensions. We also extended *Title* to 4 boolean subspace dimensions, "*Database*", "*Data Mining*", "*Information Retrieval*", and "*Machine Learning*", so totally there are 6 subspace dimensions. Since *Title* is a text field and each title has multiple keywords, we manually constructed a keyword-to-category mapping so that each paper entry is mapped to its corresponding research area. For example, a base tuple containing "XML" in the

title is mapped to *Database*, and the corresponding boolean dimension will be set to "true". A base tuple with title containing "bayesian" would have its *"Machine Learning"* dimension set to "true". The selectivities of the "true" value of the 4 boolean dimensions are between $72K$ and $226K$. To compute the aggregate scores and derive object rankings, we used the $COUNT$ (*i.e.*, number of publications) so that an author publishing more papers will be ranked higher. Note that the evaluation quality can be further enhanced by building a more accurate classifier of the research areas and introducing authority-based scoring function; however, these are not our main goals here.

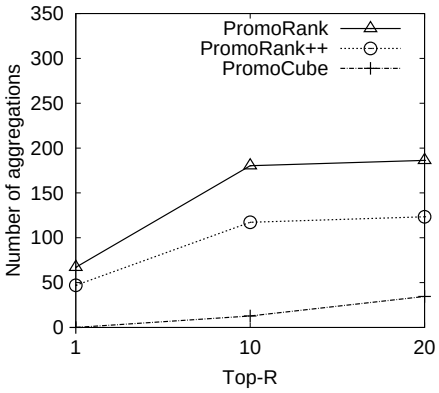
Promotiveness measure: Often users may want to enforce significance constraint on subspaces (much like an iceberg condition) rather than imposing penalty on selective subspaces. Therefore, in addition to P_1, we experimented with another measure P_2. Specifically, let P_2 be $\mathsf{Rank}^{-1} \cdot \mathcal{I}\{\mathsf{TupleCount} \geq 100\}$; that is, only subspaces having ≥ 100 base tuples (*i.e.*, a community with > 100 papers) will be considered, whereas the remaining ones will be pruned. Intuitively, this measure prevents us from searching too "deeply" in the target lattice.

Table 3.5 shows the top-3 promotive subspaces for two researchers using both P_1 and P_2. Their global ranks are also listed for comparison purposes. Not surprisingly, the subspaces characterize the authors' strengths well. We can see that P_1, which employs a penalty to smaller subspaces, prefers subspaces with large population (*i.e.*, a large $\mathsf{ObjCount}$) and reasonably high ranks. The subspace percentile ranks of the two authors are also much better than their global percentile ranks. On the other hand, by enforcing the iceberg constraint, P_2 prefers absolute high ranks as long as the subspace meets the significance constraint. Thus, the top-3 promotive subspaces by P_2 have smaller $\mathsf{ObjCount}$, but their absolute ranks are higher (*i.e.*, the two authors are consistently ranked into top-5 in the top promotive subspaces). It is worth mentioning that there is no universally best promotiveness measure, and it is up to the user to choose the proper measure according to the application.

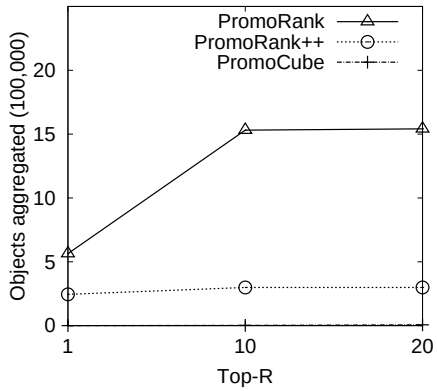
Performance: We created a workload consisting of a set of 10 randomly selected authors with ≥ 10 papers as target objects. For $\mathsf{PromoCube}$, we chose $minsig = 100$; it turns out that $K = K' = 10$ would suffice to materialize almost all distinct aggregate scores in significant subspaces since paper counts are often small integers. Thus, the resulting space overhead for the data set,



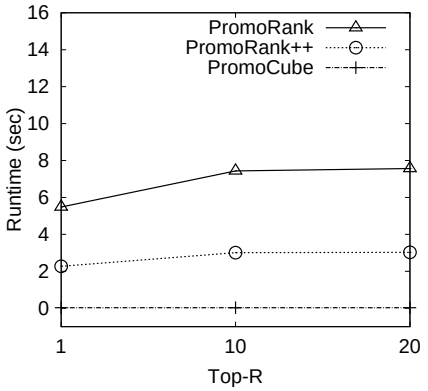
(a) Runtime vs. top-k.



(b) Number of subspace aggregations vs. top-k.



(c) Number of objects aggregated vs. top-k.



(d) Runtime on promotiveness measure P_1.

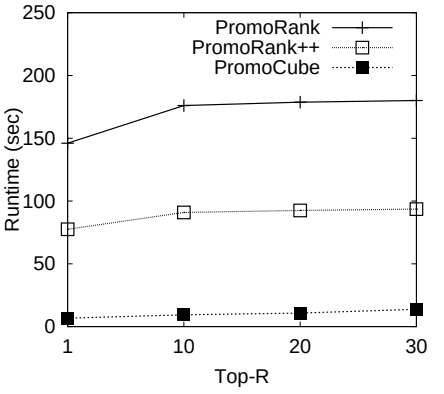
Figure 3.9: Performance results on the DBLP data.

$310KB$, can be regarded trivial, and we do not compare it with iceberg cube.

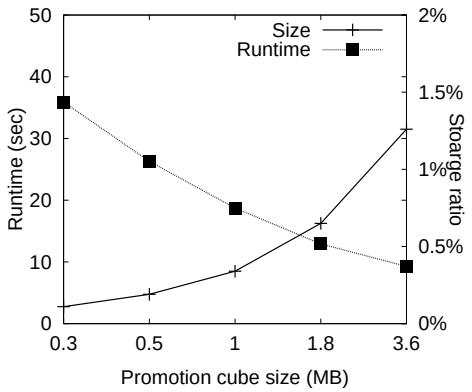
We report in Figure 3.9(a) the query execution time of the PromoRank, PromoRank++, and PromoCube algorithms when varying k, the number of subspaces to be returned. The promotiveness measure was set to P_2. We can see that all these methods become slower when k increases, as expected. PromoRank++ is consistently 2 to 2.5 times faster than PromoRank, showing the superiority of the proposed pruning techniques. PromoCube outperforms PromoRank by a ratio of 180, 9.6, and 5 when k is set to 1, 10, and 20, respectively. In particular, PromoCube performs extremely well when k is small, because in such cases, the promotion cube can directly return the result using $O(1)$ lookup time and terminate early; when k increases, additional online cost is incurred to aggregate non-pruned search space.

To explain the differences in query execution time, in Figure 3.9(b), we plot the total number of subspaces aggregated by each of the 3 algorithms with respect to k. Clearly, this figure shows that the query execution time is linearly correlated with the number of subspace aggregated. Moreover, Figure 3.9(c) shows the total number of objects aggregated by each algorithm with respect to k. Compared to PromoRank, the baseline strategy, PromoRank++ dramatically reduces the number of objects aggregated, due to the power-law distribution of author aggregate scores. In fact, PromoRank++ is able to prune out the long tail of over 50% authors with small paper count at the very initial recursion, resulting in significant cost saving of subsequent object aggregation and data partitioning.

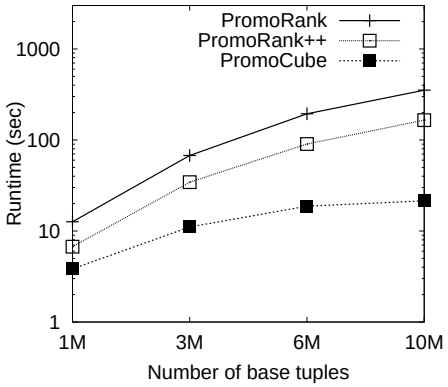
Figure 3.9(d) displays the runtime comparison of the 3 algorithms based on the promotiveness measure P_1, which enforces penalty on subspaces. Again, PromoRank++ is up to 2.5 times faster than PromoRank, verifying the effectiveness of the pruning techniques. The total number of subspaces and objects aggregated are quite similar to previous results so we do not plot them here. Moreover, PromoCube consistently performs extremely well due to the relatively small domain of aggregate scores. Actually, for all the queries being tested in here, looking up the promotion cube suffices to answer them. In other words, the target object's ranks were already precomputed in the top subspaces.



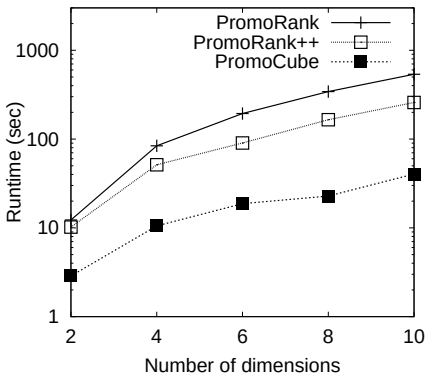
(a) Runtime vs. top-k.



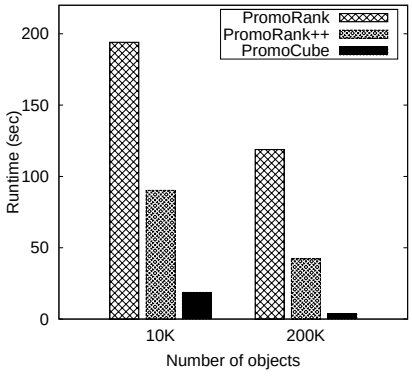
(b) Runtime vs. space overhead.



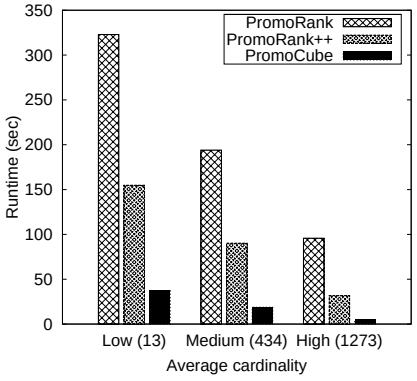
(c) Runtime vs. the number of base tuples n.



(d) Runtime vs. the number of dimensions d.



(e) Runtime vs. the number of objects.



(f) Runtime vs. average cardinality.

Figure 3.10: Performance results on the TPCH data.

3.6.4 The TPCH Data

In this subsection we report the performance results on the synthetic TPCH data [3].

Default data characteristics: We generated a default TPC-H data set with scale factor being set to 1, and the extracted the *lineitem* table containing 6,001,215 base tuples and 16 dimensions. We extracted the following 6 dimensions from the table and used them as the subspace dimensions: *l_shipdate* (2526), *l_quantity* (50), *l_discount* (11), *l_tax* (9), *l_linenumber* (7), and *l_returnflag* (3). Also, we used *l_suppkey* (10,000) as the object dimension and *l_extendedprice* (containing real numbers ranging from 901.00 to 104949.50) as the score dimension. Let SUM be the aggregate measure and objects are ranked in the descending order.

Performance: For the performance study we consider P_1 as well as a promotiveness measure P_3 with iceberg constraint TupleCount ≥ 1000. For PromoCube, we set $minsig = 1000$; for each subspace, we considered the largest $K = 1000$ aggregate scores and materialized $K' = 8$ of them at evenly spaced ranks. The resulting space overhead for PromoCube is $978.3KB$, in contrast to $277.5MB$, the size of the iceberg cube with condition TupleCount ≥ 1000 (*i.e.*, for each subspace passing the condition, materialize all aggregate scores). All results reported here were averaged over a set of 5 randomly generated target objects.

Figure 3.10(a) compares the performance of PromoRank, PromoRank++, and PromoCube in terms of k (using P_1). PromoRank++ is approximately 2 times faster than PromoRank, while PromoCube is about 20 times faster than PromoRank. Compared with DBLP, the synthetic data used here is more than 3 times larger in terms of the number of base tuples, so more query execution time is consumed by all methods. However, even when $k = 30$, PromoCube needs only 13.7 seconds on average to execute a query. This performance can be further improved when increasing the storage budget. This shows that conducting promotion analysis over large data sets is feasible with some precomputation.

Figure 3.10(b) displays the relation between promotion cube size and query execution time when fixing $k = 10$ (using P_3). To generate promotion cubes with different sizes, we fix the *minsig* parameter to 1000 and K to 1000 but vary K'. Note that the larger K' is, the more space the promotion cube will use. The sizes of the resulting five cubes range from $0.3MB$ to $3.6MB$, which are only 0.11% to 1.26% of the corresponding iceberg cube's size, $277.5MB$. We can see that

an exponentially decreasing cube size leads to only linearly increasing query execution time. In particular, when the cube size is only $0.3MB$ (0.11% of the iceberg cube size), the average query execution time is 36 seconds, about 4.8 times faster than the baseline (176 seconds as shown in Figure 3.10(a)). Note that the promotion cube is able to support any query regardless of the query. A target objects' top-k promotive subspace can be correctly computed no matter it has been precomputed or not.

Now we fix the query parameters (*i.e.*, the promotiveness measure, the set of 5 target objects, and $k = 10$), and further generate different synthetic data sets by varying each of the following parameters: (i) the number of base tuples, (ii) the number of subspace dimensions, (iii) the number of objects, and (iv) the average cardinality of subspace dimensions. We report the performance results of the algorithms as follows.

First, we evaluate the algorithms on four data sets with $1M$, $3M$, $6M$, and $10M$ base tuples. We set the space overhead of PromoCube to be proportional to the number of base tuples. The query execution time of PromoRank, PromoRank++, and PromoCube with respect to the base tuple number n is depicted in Figure 3.10(c). As expected, PromoRank++ is about 2 times faster than PromoRank, because both partitioning and aggregation costs are linear to n. On the other hand, PromoCube is increasingly faster with respect to n. This is because the PromoCube prunes subspaces before any online aggregation happens, which is unlike PromoRank++ that prunes subspaces during the online aggregation process. Therefore, PromoCube is able to avoid aggregating many subspaces before scanning the data, so the actual aggregation and partitioning cost saving of it is much larger than that of the online pruning techniques.

Second, we vary the number of subspace dimensions, d, for the default data set with $6M$ base tuples. In addition to the 6 subspace dimensions mentioned earlier, 4 more dimensions were included for this test case: *l_commitdate* (2466), *l_linestatus* (9), *l_shipmode* (7), and *l_shipinstruct* (4). As shown in Figure 3.10(d), we vary d from 2 to 10 and compare the query execution time of the algorithms using the default parameters and query workload. We can see that the gap between PromoRank++ and PromoRank is not large when $d \leq 4$. This is because the total number of target subspaces itself is quite small, so the pruning techniques would not be effective due to the overhead. When $d \geq 6$, the number of target subspaces becomes larger, so more interdependent

relationships can be exploited. Our conclusion is that increasing the number of subspace dimensions may not result in an increasing gain of PromoRank++ over PromoRank; nevertheless, it does make PromoCube more efficient (relatively) because more likely there will be some subspaces where a target object ranks high.

Third, to test the relation between the total number of objects $|\mathcal{O}|$ and query execution time, we replace *l_suppkey* (10,000) with *l_partkey* (200,000) as the object dimension on the default data set. As shown in Figure 3.10(e), when $|\mathcal{O}| = 200K$, all algorithms' absolute query execution time becomes smaller than when $|\mathcal{O}| = 10K$, because the more objects, the less the number of target subspaces there will be (*i.e.*, each object will appear in less base tuples). On the other hand, the speedup ratio of PromoRank++ goes from 2 for $10K$ objects to 2.8 for $200K$ objects. Intuitively, more objects tend to make the subspace aggregation cost outweigh the partitioning cost relatively, which would magnify the effectiveness of subspace pruning, which aims to reduce the number of subspace aggregations. For the same reason, the PromoCube algorithm goes from 10 times faster for $10K$ objects to 30 times for $200K$ objects, because the overhead of computing the target object's aggregate scores in the target lattice becomes smaller.

Lastly, we vary average cardinality by selecting 3 different sets of 6 subspace dimensions from the default $6M$ data set. We label these dimension sets as "low" (the average cardinality of the dimensions in the set is 13), "medium" (*i.e.*, the default set of 6 subspace dimensions studied earlier with average cardinality 434), and "high" (the average cardinality is 1273). As displayed in Figure 3.10(f), PromoRank++ and PromoCube are 3 and 20 times faster than PromoRank on the high-cardinality data set, and 2.1 and 8.7 times faster on the low-cardinality data set. It is found that both algorithms favor large cardinalities, where aggregates become very sparse. In such cases, the aggregate scores would be equal (*i.e.*, to base scores) across parent-child subspaces, thereby providing a tighter lower bound for rank.

3.7 Summary

In this chapter, we studied the promotion analysis problem over categorical subspaces. A unified promotiveness measure that combines both object ranking and subspace significant was introduced to quantify the promotional value of subspaces. For efficient query processing, an algorithm

framework based on recursive data partitioning and aggregation was proposed, and new pruning techniques as well as the promotion cube approach were integrated into the framework. Moreover, we identified the spurious promotion problem and used the ANOVA method to remove spurious dimensions. Our comprehensive experiments verified that the promotion query is able to discover meaningful results, and the efficiency of our proposed algorithms significantly outperform the baseline solutions.

Chapter 4

Region-based Online Promotion Analysis

4.1 Introduction

In many applications, it is important to promote an object on continuous, ranged dimensions in additional to categorical dimensions. This chapter examines the *top-k region-based promotion query* (RepQuery), the goal of which can be intuitively stated as follows: given an object of interest, such as a product or a person, we would like to discover the *top-k promotion regions* to promote the given object. Here a *region* is defined as a continuous interval or block over one or a few OLAP range dimensions, and a *promotion region* intuitively refers to a region where the given object is highly ranked among all other objects.

Example 9 *In the DBLP database, one may be interested in finding the best promotion region for a given author. While describing her as the 300-th most prolific author could be less interesting, using* RepQuery*, however, one may find her to be the most prominent author in* {Year=*2000*∼*2009*} *in the database field.*

This example illustrates the related application scenarios, where RepQuery discovers promotion regions by drilling down to different parts of the data and surfacing the regions where the given object is highly ranked (note that most objects are not highly ranked in the global region). In business intelligence applications like marketing, RepQuery is able to help users to quickly explore and understand which regions are the most likely to promote some specified product or business object among a large number of regions, and then these promotion regions along with the ranked results can serve the purpose of decision making. In particular, such region-based ranked results can help, for example, (1) analyze the best market segments (*e.g.*, a particular customer space or a geographical area) for resource allocation and promotion; (2) advertise and enhance

brand image (*e.g.*, a bestselling car model belonging to a certain price range or time frame); and, (3) discover and summarize interesting object features in not only categorical but also numerical feature spaces (*e.g.*, a highest-rated apartment rental business with no more than fifty employees).

In comparison to the basic promotion query problem that is formulated over the categorical, multidimensional space, RepQuery introduces 3 major new challenges.

First, using simple object ranking to measure top-k promotion regions could be insufficient in many cases because different regions may not be equally interesting; specifically, they may have (1) dramatically different sizes so that smaller regions should not be weighed equally as larger ones, and (2) containment relationships or overlaps that may cause redundancy in top-k results. For example, if the query object is highly ranked a region, then it may not be desirable to return another region that is enclosed in it. To this end, the semantics of the query model must incorporate *ad-hoc rank-independent weights* for regions, such that users can impose different weights on regions based on their prior knowledge. Also, the *redundancy-aware* semantics should be supported such that the top-k regions generated are discriminative, *i.e.*, no pair of top-k regions is more similar than a user-specified threshold.

Second, because RepQuery need to handle continuous, ranged dimensions whereas the previous problem deals with categorical dimensions, the search space of RepQuery would be *significantly larger*. For example, a *Year* dimension with 50 distinct values may generate only 50 subspaces but $50 \times (50 + 1)/2 = 1275$ one-dimensional regions. We can see that if there are d range dimensions, each having cardinality N, the total number of regions would be $(\frac{N(N+1)}{2})^d$, quadratic of the number of subspaces. The huge search space would dwarf the cost saving of any online pruning method, making the proposed query processing techniques in the previous chapter simply infeasible on even moderately large data set.

Third, for RepQuery, we have to tackle the *non-monotonicity* property of the aggregate measure when computing object rankings. Indeed, the previous chapter assumes that the aggregate measure for ranking be monotone such as *SUM* for the purpose of shared computation and measure bounding. The aggregate measures supported in this work can, nevertheless, be arbitrarily complex, ad-hoc measures defined by users.

Despite that the need for RepQuery is commonplace, no existing work in the literature attempts to address all these challenges. What happens to users of a conventional database system is that they would need to go through a trial-and-error process to manually search for interesting promotion regions, meaning that they have to rely heavily on prior knowledge. The results obtained in this way could be rather incomplete or even misleading. On the other hand, among the numerous database top-k query processing techniques, *none* can be applied toward solving RepQuery, because they require that a region and the number of objects displayed be specified as parameters.

Obviously, a naive implementation that iteratively aggregates each region and obtains the object's rank can be intolerable to users performing explorative analysis because of the exponential number of regions. At the other end of the spectrum, a full materialization approach would also be extremely costly even on a data set with moderate size. Thus, to efficiently answer top-k region-based promotion queries, in this chapter we propose a novel, principled framework called the *Region-based Promotion Cube (RepCube)*, grounded on a *partial materialization strategy with solid theoretical analysis*.

A key ingredient of the framework is a model of materialized cube cell's *pruning power*, which lays the foundation for an overall *cost model* that computes the pruning power of any cube structure. In the RepCube, the cell structure is similar to quantiles for estimating probability distributions; here we exploit each cell's capability in upper- and lower-bounding object ranks and hence pruning uninteresting regions at an early stage. However, unlike any previous work, our key observation is that a *uniform cell structure bookkeeping scores from evenly spaced rank positions does not generate satisfactory pruning power*. This is due to a unique property of our problem: an object is likely to be highly ranked in its top-k promotion regions. Thus, we present a cost model and its solution to generate an *optimized* cell structure adaptive to query distribution. These optimized cells are able to yield a *provably optimal* expected query execution cost.

Another idea we explore is to condense regions sharing similar aggregate score distributions. For example, the sales of products might be similar for regions $\{Year = 2007 \sim 2009\}$ and $\{Year = 2008 \sim 2010\}$ after proper normalization. Thus, we select a few *relaxed cells* to represent score ranges instead of exact scores, and use them to summarize sets of original cube cells. This would lead to a further space saving. The effectiveness of a relaxed cell's pruning power can be controlled

A	B	T (Object)	M (Measure)
a_1	b_1	t_1	0.7
a_1	b_2	t_2	0.8
a_1	b_2	t_3	0.8
a_1	b_2	t_1	0.2
a_1	b_3	t_4	1.2
a_2	b_1	t_3	0.9
a_2	b_2	t_1	0.3
a_2	b_4	t_2	1.6
a_2	b_5	t_1	1.2

Table 4.1: An example fact table.

by a user-specified parameter ϵ. In summary, our contributions are the following:

- (Section 4.2) Present the class of *top-k region-based promotion queries* and the model and semantics;
- (Section 4.3) Introduce the generic *region-based promotion cube* framework that can achieve a desired tradeoff between storage space and query execution time;
- (Section 4.4) Propose a *cost model* and a *provably optimal solution* for generating the most cost-effective cell structure through a solid theoretical analysis;
- (Section 4.5) Develop a *cell relaxation* approach to further optimize the storage overhead; and
- (Section 4.6) Present comprehensive experimental evaluation on real and synthetic data sets to verify that our framework is *1 to 2 orders of magnitude faster* than baseline solutions.

In addition, Section 4.7 discusses the related work, and Section 4.8 concludes this chapter.

4.2 Model and Semantics

In this section we present our data model and formalize the query semantics.

Data model: Consider a data set $FactTable(\mathcal{A}, \mathcal{B}, T, M)$ consisting of base tuples with the following dimensions.

R (Region)	$F(\tau, R)$	$Rank(\tau, R)$	$PRank(\tau, R)$
$R_1 : \{a_1, b_1 \sim b_1\}$	0.7	1	100%
$R_2 : \{a_1, b_1 \sim b_2\}$	0.9	1	33%
$R_3 : \{a_1, b_1 \sim b_3\}$	0.9	2	50%
$R_4 : \{a_2, b_1 \sim b_4\}$	0.3	3	100%
$R_5 : \{a_2, b_1 \sim b_5\}$	1.5	2	67%
other regions omitted...	...	...	...

Table 4.2: Example regions and an object of interest $\tau = t_1$'s aggregate score (SUM), rank, and percentile rank in each region.

- α **categorical dimensions** $\mathcal{A} = \{A_1, A_2, \ldots, A_\alpha\}$: for each $A_i \in \mathcal{A}$, $dom(A_i)$ is a finite collection of categorical values;

- β **continuous ranged dimensions** $\mathcal{B} = \{B_1, B_2, \ldots, B_\beta\}$: these are discretized numeric dimensions. For each $B_i \in \mathcal{B}$, $dom(B_j)$ consists of an ordered set of ranges of values. A typical example is $dom(\mathit{Year}) = \{2009, 2008, \ldots\}$;

- **Object dimension** T and **measure dimension** M: $dom(T)$ is the collection of **objects** and let n be the total number of distinct objects, *i.e.*, $n = |T|$. Let $dom(M)$ be real numbers $\mathbb{R}$.

A **region** R is defined as $\{a_1, \ldots, a_\alpha, b_1 \sim b'_1, \ldots, b_\beta \sim b'_\beta\}$, where $a_i \in dom(A_i)$ or $a_i =$ " $*$ " (the "don't care" value) and $b_j, b'_j \in dom(B_j)$ and $b_j \leq b'_j$. Denote by $\mathcal{R}$ the set of **all regions** in the data set (we refer readers to [17, 37, 41] for a detailed complexity analysis of $\mathcal{R}$).

Example 10 *Table 4.1 displays a sample fact table consisting of a categorical dimension A and a range dimension B. For dimension B, we assume that there is a total ordering among all the values, i.e., $b_1 < b_2 < b_3 < b_4 < b_5$. T and M are the object and measure dimensions, respectively. There are 4 objects in this sample data set. In Table 4.2, the first column R displays several example regions generated from the sample data in Table 4.1. For instance, $\{a_1, b_1 \sim b_3\}$ represents the region "$A = a_1 \wedge b_1 \leq B \leq b_3$" (i.e., all the base tuples falling in this region). For clarity of presentation, other regions now shown in this table are omitted.*

Query model: Consider an arbitrary aggregate function F (*e.g.*, *SUM*, *Average*, *Variance*). Given any region $R \in \mathcal{R}$ and any object $t \in T$, denote by $F(t, R)$ the **aggregate score** of t in region

R. Similarly, denote by $Rank(t, R)$ the object t's **rank** in R, obtained by ordering objects in the region *descendingly* or *ascendingly* according to their aggregate scores. For simplicity, throughout the chapter we assume that objects are ordered descendingly. Given the notion of object rank, the RepQuery problem can be defined as follows.

Definition 8 (Top-k Region-based Promotion Query) *Given a query denoted by $Q(\tau, k)$ consisting of an* object of interest $\tau \in dom(T)$ *for promotion and a non-negative integer k, return $\mathcal{P}$, the ordered list of top-k regions, such that for $\forall R_1 \in \mathcal{R} - \mathcal{P}, \forall R_2 \in \mathcal{P}$ we have* $\text{Rank}(\tau, R_1) \times w(R_1) \geq \text{Rank}(\tau, R_2) \times w(R_2)$*, where $w(\cdot)$ is any non-negative weight function over $\mathcal{R}$.*

Ties are broken arbitrarily. $w(\cdot)$ is a weight function that assigns a non-negative weight for each region so that regions can have different importance scores. A larger weight of a region indicates that the region is less important. Note that this weight function is different from the "subspace significance" concept introduced in the previous chapter, because we allow it to be ad-hoc. We do not put $w(\cdot)$ as a query parameter for clarity of presentation. We can see that, by setting $w(\cdot)$ to a positive constant, the RepQuery model admits a simple object ranking semantics in that it asks for the top regions where the given object τ is highly ranked. A user may further model **percentile rank** (*PRank*) by letting $w(R)$ be the inverse of the number of objects present in region R. One may also let $w(R)$ be the inverse of R's number of tuples to **discount small regions**. Note that these rank-independent weights can be specified by users or combined in an ad-hoc way to tackle more complex scenarios.

Example 11 *Continuing from the last example, let the object of interest τ be t_1. Table 4.2 shows τ's aggregate score $F(\tau, R)$ (using SUM), rank* $\text{Rank}(\tau, R)$*, and percentile rank* $\text{PRank}(\tau, R)$ *in the example regions. Not counting the omitted regions, R_1 and R_2 would be the top-2 promotion regions using simple ranking (because t_1 is ranked top-1 in both regions), whereas R_2 and R_3 would be the top-2 promotion regions according to* PRank *(because t_1 is ranked top-33% and top-50% in them, respectively).*

In some cases, however, the top-k regions produced by the above definition may contain many overlaps. It is often desirable to remove the redundancy from the results such that the top-k regions do not contain redundancy. For example, if the object of interest is highly ranked in two neighbor

regions $\{2000 \sim 2009\}$ and $\{2001 \sim 2010\}$, it would be desirable to output only one of them. For another example, when the object of interest is highly ranked in a large region, then it might not be necessary to output any smaller region enclosed in the large one. Motivated by this, we further describe a query model that incorporates the redundancy-aware semantics.

Definition 9 (Top-k Discriminative Region-based Promotion Query) *Given a query denoted by $Q(\tau, k, \theta)$, return $\mathcal{P}$, the ordered list of* top-k discriminative regions*, such that for $\forall R_1 \in \mathcal{R} - \mathcal{P}$, we have either*

$$\forall R_2 \in \mathcal{P} \Rightarrow \text{Rank}(\tau, R_1) \times w(R_1) \geq \text{Rank}(\tau, R_2) \times w(R_2),$$

or

$$\exists R_2 \in \mathcal{P} \Rightarrow \text{Sim}(R_1, R_2) \geq \theta \wedge \text{Rank}(\tau, R_1) \times w(R_1) \geq \text{Rank}(\tau, R_2) \times w(R_2).$$

We elaborate on this definition. First, to gauge the similarity between two regions, we use $Sim(R_1, R_2) = \frac{|R_1 \cap R_2|}{|R_1 \cup R_2|}$, where $|R|$ denotes the number of base tuples contained in R. Other symmetric similarity measures can also be applied in principle. For example, one may use the cosine measure or define $|R|$ as the number of objects in region R. Second, the redundancy-aware semantics can be explained as follows. For any region R_1 that is not in the top-k results, we assume that either it has a larger weighted rank (*i.e.*, $Rank \times w$) than all top-k regions, or it has a larger weighted rank than some top-k region R_2 and is redundant with R_2. This definition would guarantee that (i) the top-k regions can promote the given object well because no other k regions are better; and (ii) the top-k regions do not contain any redundancy. Clearly, this definition is useful when users want to seek more information from the top-k results. We illustrate the definition using the example below.

Example 12 *Following from the running example (Tables 4.1 and 4.2), consider a top-k discriminative query $Q(t_1, 2, 0.6)$, that is, the object of interest is t_1, $k = 2$, and the similarity threshold θ is set to 0.6. We can see that R_2 and R_3 have a similarity value* $\text{Sim}(R_2, R_3) = |\{a_1, b_1 \sim b_2\}|/|\{a_1, b_1 \sim b_3\}| = 4/5 = 0.8 > \theta = 0.6$. *Thus, based on* PRank*, R_3 is no longer a top-2 promotion region since it is redundant with R_2, which has a better* PRank *than P_3; instead, now*

the top-2 discriminative promotion regions should be R_2 and R_5, which are not redundant (again, we do not count the omitted regions in Table 4.2).

Before presenting our solutions, assume that the aggregate function F is fixed. For clarify of presentation, also assume Definition 1 is used and let $w(\cdot)$ be 1 (*i.e.*, simple object ranking). More complex semantics will be addressed in Section 4.5.

4.3 RepCube: The Region-based Promotion Cube Framework

In this section we first motivate and present the RepCube framework (Sections 4.3.1, 4.3.2, and 4.3.3) and then present the RepQuery execution algorithm as an integral part of the framework (Section 4.3.4). This framework lays the foundation for the subsequent structure optimization techniques (Sections 4.4 and 4.5).

4.3.1 No Materialization and the GetRank Primitive

Let us first consider a no-materialization strategy. In this case, a promotion query must be computed from scratch. The basic query execution method is to enumerate each region $R_i \in \mathcal{R}$ ($1 \leq i \leq |\mathcal{R}|$) and compute $Rank(\tau, R_i)$; the top-k promotion regions are maintained and outputted. During query execution, we abstract out a data access primitive `GetRank`(i), for computing $Rank(\tau, R_i)$, which can be implemented using the following SQL statement:

`select` Rank(τ, R_i)
`from` *FactTable*
`where` region R_i
`group by` object dimension T
`order by` F(*Measure*) `desc`.

`GetRank`(i) accesses all base tuples in R_i, computes aggregate scores for all objects, and derives τ's rank. `GetRank`() is an expensive operation due to its *holistic* property: all objects must be aggregated to correctly compute τ's rank in the region. This means that all base tuples in R_i need to be accessed and aggregated. Since the selectivity of a region could be large, and there could

also be a large number of regions, the on-the-fly query execution algorithm would be extremely expensive even with some aggregation cost sharing and pruning techniques.

4.3.2 Full Materialization and the GetAgg Primitive

On the other extreme, a full-materialization approach means to precompute all object aggregate scores for all regions. During query execution, we abstract out another data access primitive `GetAgg`(i), which computes $F(\tau, R_i)$ as follows: simply retrieves all base tuples in R_i related to τ and aggregates them. The SQL implementation can be the following:

`select` $F(\tau, R_i)$
`from` *FactTable*
`where` region R_i `and` $T = \tau$.

Given a full materialization, a query can be executed in 2 steps for each of the $|\mathcal{R}|$ regions: first, call `GetAgg`(i) to get $F(\tau, R_i)$, and second, derive $Rank(\tau, R_i)$ by counting the materialized aggregate scores in R_i greater than $F(\tau, R_i)$. The top-k answers can be subsequently computed. Compared to `GetRank`(), `GetAgg`() is much less costly because it only accesses τ's base tuples in region R_i, so the selectivity of the primitive is very high. Also, no group-by operation is needed here. `GetAgg`() can be made even more efficient by constructing a clustered index on T.

Not surprisingly, the storage overhead would be prohibitive. For example, if a data set has 1 categorical dimension and 2 range dimensions with an average cardinality of 100 as well as $10K$ objects, the full materialization approach would approximately generate $101*(100*101/2)^2*10K \approx 2.6*10^{13}$ values, or, equivalently, nearly $100TB$ of disk space for a single aggregate function!

4.3.3 The Uniform RepCube Structure

To balance the storage overhead and online execution time, we use a method similar to quantization that reduces the storage cost while facilitating online pruning. This method samples aggregate scores at predefined positions from a sorted list of aggregate scores for each region R_i. Thus, each region will have some sampled aggregate scores. We call the materialized sample scores for each region a **p-cell**, formally defined as follows.

Definition 10 (P-Cell) *For any region $R_i \in \mathcal{R}$, denote by $F_1^i, F_2^i, \ldots, F_n^i$ the complete ranked list of object aggregate scores in R_i (without loss of generality, assume decreasing order and no duplicate scores). Given a* ***position vector*** *of length m: $\vec{\phi} = (\phi_j)_{j=1}^m$ (where $0 \leq m \leq n$, $1 \leq \phi_j \leq n$, and $j < l \Rightarrow \phi_j < \phi_l$) , define $PCell^i$ as the vector of aggregate scores induced by $\vec{\phi}$, i.e., $PCell^i = (F_{\phi_j}^i)_{j=1}^m$.*

The position vector will uniquely determine the content of the p-cells. Suppose m, the length of the position vector, is given, the most common way of choosing the position vector is to select a collection of evenly spaced values from $\{1, 2, \ldots, n\}$. A materialization plan consisting of a collection of p-cells based on such a position vector is called a *uniform RepCube*.

Definition 11 (Uniform RepCube) *A uniform region-based promotion cube is defined as a collection of p-cells, $\{R_i, PCell^i \big| 1 \leq i \leq |\mathcal{R}|\}$, where the position vector is set to $\vec{\phi} = (1 + \lfloor \frac{j-1}{m} \times n \rfloor)_{j=1}^m$.*

Clearly, m controls the size of the uniform RepCube. Let us first look at two extreme cases. First, $m = 0$ corresponds to the no materialization strategy, since the position vector is empty and no aggregate score will be materialized for any region; when $m = n$, the position vector must be $(12 \cdot n)$ so that all object aggregate scores are materialized for each region. This is equivalent to the full materialization strategy. In effect, m is much smaller than the total number of objects n so that the uniform RepCube would be significantly smaller compared to a full-materialization approach.

4.3.4 Query Execution Algorithm

Let us describe the query execution algorithm given an object of interest τ and the uniform RepCube structure. Recall that its goal is to return the top-k regions where τ is the most highly ranked. The query execution works in 2 phases. First is a **pruning phase**, where upper and lower bound ranks of τ for each region can be computed using the uniform RepCube. Then the unpromising regions not possible to be in the top-k are pruned. The second is a **verification phase** where each of the potential top-k regions is verified such that τ's true rank can be computed.

Both phases can be succinctly represented using the `GetRank`() and `GetAgg`() primitives. The detailed algorithm is depicted in Table 4.3 and we elaborate on each step. The pruning phase

Algorithm 4.1: Query Execution

```
    /* Pruning phase */
1:  for i ← 1 to |R| do
2:      F(τ, R_i) ← GetAgg(i); /* not costly */
3:      LBRank_i ← φ_j + 1, where j satisfies
            PCell^i_j > F(τ, R_i) ≥ PCell^i_{j+1};
4:      UBRank_i ← φ_l − 1, where l satisfies
            PCell^i_{l−1} ≥ F(τ, R_i) > PCell^i_l;
5:  end
6:  δ = the k-th smallest UBRank_i for 1 ≤ i ≤ |R|;
7:  R* ← {R_i | LBRank_i ≤ δ}; /* unpruned set of regions */
    /* Verification phase */
8:  foreach unpruned region R_i ∈ R* do
9:      Rank(τ, R_i) ← GetRank(i); /* costly */
10: end
11: Return P, the top-k regions with the smallest Rank(τ, R_i);
```

Table 4.3: The complete query execution algorithm.

computes the lower and upper bound ranks of τ for each region R_i and conduct pruning (Lines 1–7). Specifically, `GetAgg`(i) is called to get τ's aggregate score in region R_i (Line 2), which is subsequently compared to the region's materialized p-cell to obtain the highest possible rank $LBRank_i$ (Line 3) and the lowest possible rank $UBRank_i$ (Line 4) of τ in R_i (for correctness we define two dummy positions $\phi_0 = 0$ and $\phi_{m+1} = n + 1$ such that $PCell^i_0 = -\infty$ and $PCell^i_{m+1} = +\infty$). Next, δ is computed as a threshold, meaning that τ must rank no lower than δ in any top-k promotion region (Line 6). All regions with the best possible rank lower than δ must be unpromising and can be safely pruned (Line 7). During the second phase, we verify the unpruned regions (Lines 8–10). The `GetRank`() primitive is called for obtaining the exact rank for each unpruned region (Line 9). After obtaining the exact ranks for all promising regions, we finally compute and output the top-k promotion regions (Line 11).

Cost analysis: Since the costly `GetRank`() method (Line 9) accounts for the bottleneck of the algorithm, the cost of the query execution algorithm is dictated by $|\mathcal{R}^*|$ (as shown in Table 4.3, $\mathcal{R}^*$ represents the set of promising regions), the number of `GetRank`() calls. The number of `GetRank`() calls is in turn determined by the underlying pruning power of the materialized cube: the larger pruning power, the less calls one will need. For the uniform RepCube, the pruning power is positively correlated with the user-specified parameter m. That is, when $m = 0$ (no

materialization) no region would be pruned; when $m = n$ (full materialization) all regions will be pruned and no `GetRank`() call is needed since *LBRank* and *UBRank* are tight for all regions in this case. Therefore, the RepCube framework offers a controllable tradeoff between storage space and online execution cost.

4.4 Pruning Power Optimization for RepCube

The uniform RepCube strategy samples aggregate scores at regularly spaced positions; however, an important intuition it fails to model is that typically in top-k promotion regions τ is very likely to be highly ranked. Intuitively, selecting the uniform position vector may not be the best solution for answering region-based promotion queries, because k is small for many queries and it would be a better idea to store more samples toward highly ranked positions in order to better bound the ranks. This motivates us to investigate how to select a position vector so as to achieve the best tradeoff between the online and offline costs.

Our idea here is to carefully select a position vector *adaptive to the underlying distribution of queries* in order to achieve much better pruning power given a limited amount of storage space. Therefore, we will model the goodness of a position vector in terms of its pruning power. However, given a position vector length m, there are $\binom{n}{m}$ possible position vectors, and it would be impossible to enumerate each one and measure the online cost. To avoid exhaustive enumeration, we present an efficient and optimal algorithm.

We begin with by stating the optimization problem that we aim at solving in this section.

Definition 12 (The Pruning Power Optimization Problem) *Given a limited space budget indicated by m, and a distribution of promotion queries, determine the best position vector $\vec{\phi}$ such that the expected promotion query execution cost is minimized.*

To solve the problem, we first formulate a cost model to compute the expected RepQuery cost as a function of the position vector (Sections 4.4.1 and 4.4.2). We then discuss a dynamic programming solution for selecting the positive vector that produces the provably maximum pruning power (Section 4.4.3).

Query execution	Storage overhead	Number of `GetRank()` calls
No materialization	0	$\lvert\mathcal{R}\rvert$
Full materialization	Prohibitive	0
Uniform RepCube	Small	$< \lvert\mathcal{R}\rvert$ ($\vec{\phi}$ not optimized)
Optimal RepCube	Small	$\ll \lvert\mathcal{R}\rvert$ ($\vec{\phi}$ optimized)
Relaxed RepCube	Very small	Opt. with ϵ-relaxation bound

Table 4.4: A roadmap of different strategies studied in this chapter.

Roadmap: Table 4.4 presents a summary of the methods studied in the chapter. In the previous section we have explained the first 3 methods, namely on-the-fly execution, naive precomputation, and the uniform RepCube approaches. We can see from the table that the no materialization strategy incurs the maximum number of `GetRank()` calls, whereas the full materialization strategy incurs prohibitive storage cost. The uniform RepCube structure is able to achieve some balance between space and time, but this tradeoff is moderate. The optimal RepCube approach discussed in this section will further enhance the query efficiency as a result of a much smaller number of calls to `GetRank()`. We will prove that the optimal approach can minimize the number of `GetRank()` calls given a storage budget. Furthermore, Section 4.5 will discuss the *Relaxed RepCube* technique for reducing the storage space of the optimal RepCube.

4.4.1 The Unit Cost Model

As a building block to the overall cost model, we would like to model the basic case, that is, the cost of a single fixed RepQuery $Q(\tau_0, k_0)$ given a position vector $\vec{\phi}$ with length 1 (*i.e.*, $m = 1$); in other words, only a single aggregate score sample is drawn for each region.

Let $R_{s_1}, R_{s_2}, \ldots, R_{s_{|\mathcal{R}|}}$ be the ordered list of regions sorted according to τ_0's rank, where $s_1, s_2, \ldots, s_{|\mathcal{R}|}$ is a permutation of $1, 2, \ldots, |\mathcal{R}|$ and $i < j \Rightarrow Rank(\tau_0, R_{s_i}) \leq Rank(\tau_0, R_{s_j})$. For ease of exposition we assume τ occurs in all regions and use a short notation $Rank(i)$ to denote $Rank(\tau_0, R_{s_i})$. Since $m = 1$, let $\vec{\phi}$ be a scalar ϕ_1 ($\in \{1, 2, \ldots, n\}$), and assume that ϕ_1's corresponding p-cells, $\{PCell^{s_1}, PCell^{s_2}, \ldots, PCell^{s_{|\mathcal{R}|}}\}$, have been precomputed.

Given these p-cells, let us hypothetically compute the rank bounds in the query execution algorithm. For $Q(\tau_0, k_0)$, the computation can be divided into two cases: (1) all regions $\{R_{s_i}\}$ satisfying $Rank(i) < \phi_1$ will have $LBRank_{s_i} = 1$ and $UBRank_{s_i} = \phi_1 - 1$ because the inequality

$+\infty = PCell_0^{s_i} < F(\tau_0, k_0) < PCell_1^{s_i}$ holds; (2) conversely, all regions $\{R_{s_i}\}$ satisfying $Rank(i) > \phi_1$ will have $LBRank_{s_i} = \phi_1 + 1$ and $UBRank_{s_i} = n$. Without loss of generality, assume that there does not exist any region R_{s_i} such that $Rank(i) = \phi_1$ (any region satisfying this equation cannot be pruned since its *LBRank* would be 1 and *UBRank* would be n). Now, define i^* to be the value in $\{1, 2, \ldots, |\mathcal{R}|\}$ such that $Rank(i^*) < \phi_1 < Rank(i^* + 1)$ (let $Rank(|\mathcal{R}| + 1)$ be ∞). Based on i^*, we can precisely compute $\mathcal{R}^*$, the unpruned set of regions for $Q(\tau_0, k_0)$, as in either of the following cases:

- When $i^* < k_0$, since in this case there are less than k_0 regions with *UBRank* equal to $\phi_1 - 1$, the k_0-th *UBRank* would be n, meaning that $\delta = n$ (*i.e.*, the rank threshold δ does not have any pruning power because the query object will be ranked higher than n-th in any region). Hence, no region can be pruned and the set of promising regions is equal to the set of all regions, *i.e.*, $\mathcal{R}^* = \mathcal{R}$.

- When $i^* \geq k_0$, there would be exactly i^* regions with *UBRank* equal to $\phi_1 - 1$ and *LBRank* equal to 1. For the remaining $|\mathcal{R}| - i^*$ regions, their *LBRank* will be $\phi_1 + 1$. Thus, the rank threshold δ will be $\phi_1 - 1$, and so the remaining $|\mathcal{R}| - i^*$ regions will be pruned since they have a rank no better than the threshold. Hence, $\mathcal{R}^* = \{R_{s_1}, \ldots, R_{s_{i^*}}\}$.

Example 13 *Suppose the object τ_0 is given, the query parameter k_0 is 2, and there are totally 100 objects and $|\mathcal{R}| = 6$ regions $R_1, R_2, \ldots, R_6$, in which τ_0 is ranked 38th, 35th, 26th, 41st, 29th, and 50th, respectively. Thus, $s_1 = 3, s_2 = 5, s_3 = 2, s_4 = 1, s_5 = 4$, and $s_6 = 6$ based on the ranks in these regions. Now, let us hypothetically construct a uniform RepCube using a single-length position vector (scalar) $\phi_1 = 27$ and observe how the query is executed. Based on the definition of i^*, we will obtain $i^* = 1$ because $Rank(1) < \phi_1 < Rank(2)$ (i.e., $26 < \phi_1 < 29$). During the query execution, only in R_3 does τ_0 have $UBRank = \phi_1 - 1 = 26$, whereas in all other regions τ_0 has $UBRank = n = 100$. Consequently, δ, the k_0-th smallest UBRank, will be 100 and therefore none of the 6 regions can be pruned after the pruning phase. In other words, this uniform RepCube does not help this query at all.*

Now, if we construct another uniform RepCube by setting ϕ_1 to 37 and simulate the query execution process, we will obtain $i^* = 3$ since $Rank(3) < \phi_1 < Rank(4)$. Thus R_3, R_5, and R_2 have

UBRank $= 36$. *In this case* $\delta = 36$ *and the remaining 3 regions can be pruned. Therefore, the uniform RepCube constructed in this way would improve the performance of the given query.*

Now we are ready to present the unit cost model. We introduce some notation. Denote by

- $COST(Q|\phi_1)$ the overall query execution cost for $Q(\tau_0, k_0)$ given ϕ_1;
- Ω the constant cost of the pruning phase (Lines 1–7, Table 4.3);
- $COST(s_i)$ the cost of calling the `GetRank`(s_i) method for region R_{s_i}.

Since the overall query execution cost given ϕ_1 can be broken down to (1) the constant cost of the pruning phase and (2) the cost of the verification phase that consists of multiple `GetRank`() calls, we can formulate the unit query execution cost of a single query $Q(\tau_0, k_0)$ using a 1-length position vector $\vec{\phi} = \phi_1$ as the following (note that i^* can be computed using the method described earlier in this subsection):

$$COST(Q|\phi_1) = \begin{cases} \Omega + \sum_{i=1}^{i^*} COST(s_i), & \text{if } i^* \geq k_0 \\ \Omega + \sum_{i=1}^{|\mathcal{R}|} COST(s_i), & \text{if } i^* < k_0 \end{cases}$$

4.4.2 The Complete Cost Model

Now a step further. Consider the cost of a single query $Q(\tau_0, k_0)$ when a position vector $\vec{\phi} = \{\phi_j\}_{j=1}^{m}$ with arbitrary length $m \in [1, n]$ is given. As a more general case of the unit case, our goal now is to compute $COST(Q|\vec{\phi})$, the overall cost for $Q(\tau_0, k_0)$ given $\vec{\phi}$.

Note that now in each region we materialize m aggregate scores. For each $j \in \{1, 2, \ldots, m\}$, we let i_j^* be the value in $\{1, 2, \ldots, |\mathcal{R}|\}$ which satisfies $Rank(i_j^*) < \phi_j < Rank(i_j^* + 1)$. This intuitively means that if we materialize the aggregate score at position ϕ_j, we will be able to distinguish i_j^* regions from the remaining ones in terms of rank bounds. We have $i_1^* \leq i_2^* \leq \cdots \leq i_m^*$ because of the monotonicity of $\{\phi_j\}$ (*i.e.*, $\phi_1 < \phi_2 < \ldots < \phi_m$). Using a similar method for the unit cost model, the total cost of $Q(\tau_0, k_0)$ given $\vec{\phi}$ can be computed in either one of the following cases.

- When $i_m^* < k_0$, δ would be n for the same reason as in the case of $m = 1$. Hence, no region can be pruned and $\mathcal{R}^* = \mathcal{R}$.

- Otherwise, let i^* be the smallest value in $\{i_j^*\}$ to satisfy $i^* \geq k_0$. Let u be the subscript satisfying $i_u^* = i^*$. Observe that, based on the computation of *LBRank* and *UBRank*, there are exactly i^* regions having $UBRank \leq \phi_u - 1$, and the remaining $|\mathcal{R}| - i^*$ regions having $LBRank \geq \phi_u + 1$. This means that $\delta = \phi_u - 1$ and the latter $|\mathcal{R}| - i^*$ regions will be pruned. Hence, $\mathcal{R}^* = \{R_{s_1}, \ldots, R_{s_{i^*}}\}$.

Consequently, $COST(Q|\vec{\phi})$ can be formulated as:

$$COST(Q|\vec{\phi}) = \begin{cases} \Omega + \sum_{i=1}^{|\mathcal{R}|} COST(s_i) & \text{if } i_m^* < k_0 \\ \Omega + \sum_{i=1}^{i^*} COST(s_i) & \text{otherwise} \end{cases}$$

Finally, to complete the overall cost model formulation, suppose the top-k region-based promotion queries are drawn from a multivariate distribution $Q \sim p(\tau, k)$, and denote by $COST_{all}$ the variable of the overall cost induced by $p(\tau, k)$. Then the expected overall cost for any given position vector $\vec{\phi}$ can be computed as:

$$E(COST_{all}|\vec{\phi}) = \int_Q COST(Q|\vec{\phi}) p(Q) dQ. \tag{4.1}$$

Because our goal is to decide the best position vector so as to minimize the expected overall cost, the objective of the pruning power optimization problem becomes to obtain

$$\vec{\phi}^* = \arg\min_{\vec{\phi}} E(COST_{all}|\vec{\phi}). \tag{4.2}$$

4.4.3 An Optimal Solution for Maximizing the Pruning Power

This subsection discusses an efficient dynamic programming solution to compute the optimal position vector $\vec{\phi}^*$ defined in Equation 2. The idea of the dynamic programming solution is to solve a series of recurrences represented by a matrix $MinCost[i, j]$ $(0 \leq i \leq n, 0 \leq j \leq m)$. Each element of the matrix $MinCost[i, j]$ represents the minimum expected overall cost that can be achieved when selecting a j-length position vector $\vec{\phi}$ with the very last position value being i (*i.e.*, $\phi_1 < \phi_2 < \ldots < \phi_j = i$). Corresponding to $MinCost[\cdot, \cdot]$, we use another matrix $\Phi[\cdot, \cdot]$ to remember the optimal position vector that achieves $MinCost[i, j]$, *i.e.*, $MinCost[i, j] = E(COST_{all}|\Phi[i, j])$

and the last value in vector $\Phi[i,j]$ is i. The minimum value in $MinCost[\cdot,\cdot]$ will be the optimal expected overall cost.

The set of recurrences can be computed as follows. Initially, set $MinCost[0,j] = MinCost[i,0] = +\infty$ for $0 \leq i \leq n$ and $0 \leq j \leq m$, respectively, as boundary cases; also set the corresponding $\Phi[0,j]$ and $\Phi[i,0]$ to empty vectors. This initial setting means that the cost is large when nothing is materialized. Then:

$$MinCost[i,j] = \min \begin{cases} MinCost[i,j-1]; \\ MinCost[l,j-1] - \Delta(i,j,l), \\ \qquad \text{for each } 0 \leq l < i; \end{cases}$$

$$\Phi[i,j] = \begin{cases} \Phi[i,j-1], \\ \quad \text{if } MinCost[i,j] = MinCost[i,j-1]; \\ \Phi[l,j-1] \oplus i, \\ \quad \text{if } MinCost[i,j] = MinCost[l,j-1]; \end{cases}$$

Before getting into the details of the above equations, we can see that the optimal position vector of $\Phi[i,j]$ can be derived by considering either the minimum cost of an existing solution $\Phi[i,j-1]$ (*i.e.*, the optimal $(j-1)$-length position vector by considering the last positions being i), or a set of new solutions $\Phi[l,j-1] \oplus i$ for $0 \leq l < i$ (*i.e.*, new vectors composed by appending the value i to previous solutions $\Phi[l,j-1]$). For each such new solution $\Phi[l,j-1] \oplus i$, its cost can be expressed as $MinCost[l,j-1] - \Delta(i,j,l)$, where $\Delta(i,j,l) = E(COST_{all}|\Phi[l,j-1]) - E(COST_{all}|\Phi[l,j-1] \oplus i)$, *i.e.*, the reduction in expected query execution cost.

Based on Equation 4.1, the computation of the expected overall costs $E(COST_{all}|\Phi[l,j-1])$ and $E(COST_{all}|\Phi[l,j-1] \oplus i)$ requires the knowledge about query distribution. To obtain the distribution, we may assume that a query workload W consisting of w queries $\{Q_1, Q_2, \ldots, Q_w\}$ is given. In the absence of such a query workload, one may either assume that $p(\tau,k)$ be a uniform distribution and draw sample queries from it, or use application-dependent knowledge (*e.g.*, a query

interface that returns the top-10 regions). Given such a workload, $\Delta(i, j, l)$ can be computed as

$$\begin{aligned}
& \Delta(i, j, l) \\
= \; & E(COST_{all}|\Phi[l, j-1]) - E(COST_{all}|\Phi[l, j-1] \oplus i) \\
= \; & \sum_{Q \in W} p(Q)\, COST(Q|\Phi[l, j-1]) - \sum_{Q \in W} p(Q)\, COST(Q|\Phi[l, j-1] \oplus i) \\
= \; & \sum_{Q \in W} p(Q)\big(COST(Q|\Phi[l, j-1]) - COST(Q|\Phi[l, j-1] \oplus i)\big).
\end{aligned}$$

Therefore, using the above procedure, one can iterate through each $0 \leq i \leq n$ and $0 \leq j \leq m$ and fill in the matrices *MinCost* and Φ. The recurrence with the minimum cost, $\min\{MinCost[i, j]\}$, would indicate the minimum expected overall cost.

Claim 2 (Solution Optimality) *Letting*

$$(i^{opt}, j^{opt}) = \arg\min_{(i,j)} MinCost[i, j],$$

we have

$$\vec{\phi}^* = \Phi[i^{opt}, j^{opt}],$$

where $\vec{\phi}^*$ is defined in Equation 4.2.

Proof sketch: The proof of the optimality of the dynamic programming solution follows from two properties. (1) *Monotonicity*: appending a new position value to an existing vector would never increase the overall expected cost. This is obvious as materializing more aggregate scores would not hurt the efficiency query execution. (2) *Substructure optimality*: the cost reduction by appending a new position value, $\Delta(i, j, l)$, depends only on position l but none of the positions prior to l (in other words, once we know $MinCost[l, j-1]$, the values of $MinCost[l', j-1]$ for $l' < l$ would not affect $MinCost[i, j]$). As a result, at each iteration we can guarantee that $MinCost[i, j]$ is optimal to the subproblem corresponding to that iteration. Based on these properties, we can prove that the dynamic programming equations generate the best overall solution.

4.4.4 Implementation of the Optimal RepCube

Based on the selection of optimal positions, an *optimal RepCube* can be implemented in 2 steps. First, given a query distribution or workload, compute the optimal position vector $\vec{\phi}^*$ through dynamic programming. Second, for each region, materialize its p-cell according to $\vec{\phi}^*$. In this subsection, we discuss several issues concerning the implementation.

Dynamic programming complexity: The dynamic programming algorithm can be implemented using 3 nested loops for i, j, and l, respectively. At each iteration within the loop, evaluating $\Delta(i, j, l)$ is the bottleneck because of repetitive evaluations of $COST(Q|\vec{\phi})$ (each of which requires an on-the-fly promotion query). To make it efficient, we materialize $\{Rank(\tau, R_1), \ldots, Rank(\tau, R_{|\mathcal{R}|})\}$ for each $Q \in W$ upfront. This way, $COST(Q|\vec{\phi})$ (see Section 4.2) can be evaluated efficiently in $O(m \log |\mathcal{R}|)$ time without accessing the original data set during the computation of recurrences. The complexity of the nested loops would be $O(n^2 mw)$. When n is extremely large (*e.g.*, $1M$), a heuristic is to limit the choices of positions to a regularly sampled subset of $\{1, 2, \ldots, n\}$ to narrow the search space, thereby reducing both the space and time complexity.

Cost model parameters: The assignment of the cost model's parameters Ω and $COST(s_i)$ (Sections 4.4.1 and 4.4.2) can be determined depending on the underlying database. One way is to set Ω to τ's cardinality (*i.e.*, the number of base tuples containing τ) and $COST(s_i)$ to region R_{s_i}'s number of tuples. More intricate methods can be designed if one wants to share computational costs between contiguous or overlapping regions, which are beyond the scope of this study.

RepCube materialization: The size of the optimal RepCube is $O(m|\mathcal{R}|)$, which is determined by the only parameter m, and users can specify it to obtain their desired performance. Because it would be simply impossible to write a fully-precomputed cube to disk even for moderately large data sets, the offline materialization of the optimal RepCube must be implemented in a "streaming" fashion: iteratively enumerate each region and write back to disk the aggregate scores at the optimal positions. An improvement would be to share data scanning cost among overlapping regions.

If the number of categorical and ranged dimensions is high, the total number of regions, $|\mathcal{R}|$, will be exponentially large. In such cases, however, many regions will be sparse (*i.e.*, containing very few objects) or meaningless (*i.e.*, corresponding to too many predicates) and we can ignore them during materialization.

Algorithm 4.2: Generalized Query Execution

```
    /* Line 1 is the same as Algorithm 4.1 Lines 1–5 */
1:  for i ← 1 to |R| do
2:      compute LBRank_i and UBRank_i
3:  C ← {R_1, R_2, ..., R_|R|}; /* candidate regions */
4:  P ← empty list;
5:  while |P| ≤ k ∧ |C| > 0 do
6:      R ← the region in C having the highest rank of τ;
7:      Append R to P;
8:      C ← C\{R'|Sim(R', R) ≥ θ};
9:  end
10: Return P, the top-k discriminative regions;
```

Table 4.5: Computing top-k discriminative promotion regions.

4.4.5 Extensions

In our previous discussion we have assumed that the simple object ranking semantics is used, *i.e.*, $w(\cdot) = 1$ for any region. We now discuss how we can extend those techniques for supporting ad-hoc weight function as well as the top-k discriminative query semantics.

Ad-hoc weight function: To handle RepQuery with arbitrary weight functions, the query execution algorithm can be extended with only minor modification. Specifically, in Algorithm 4.1, when computing the threshold δ (Line 6) and the unpruned set of regions $\mathcal{R}^*$ (Line 7), $LBRank_i$ and $UBRank_i$ should be replaced by $LBRank_i \times w(R_i)$ and $UBRank_i \times w(R_i)$ (Lines 3–4) respectively. Correspondingly, in the cost model formulation (Section 4.4.2), $COST(Q|\vec{\phi})$ need to compute the unpruned set of regions $\mathcal{R}^*$ in the same fashion. The position vector computation algorithm remains unchanged and its optimality still holds.

Top-k discriminative promotion regions: To compute the top-k discriminative regions according to Definition 2, we show a *generalized query execution algorithm* in Table 4.5. The initial phase of Algorithm 4.2 (Lines 1–2) is similar to Algorithm 4.1 in that the upper bound rank and lower bound rank are computed for each region using the `GetAgg()` primitive. We introduce a new variable $\mathcal{C}$ as the set of candidate regions. It is initialized to be the complete set of regions (Line 3). The top-k discriminative results, denoted by $\mathcal{P}$, is initialized to be empty (Line 4).

The pruning and verification phases of Algorithm 4.2 are different from the previous algorithm in that the top-k discriminative regions are now computed one at a time. A number of iterations are executed in the while loop (Line 5–9) until either k regions are generated or the candidate

set $\mathcal{C}$ becomes empty (Line 5). At each iteration, we generate the next top discriminative region (Line 6) as follows: (i) we set the best (smallest) *UBRank* among all candidate regions in $\mathcal{C}$ as a threshold and then call `GetRank`() for those (promising) candidate regions whose *LBRank* is no lower (larger) than the threshold to obtain their exact ranks; (ii) this way we can get the next best region R among all candidate regions. We add R to the top-k result list (Lines 7), and R's similar regions (including R itself) are subsequently removed from the candidates (Line 8) to guarantee that no top-k result will be redundant. We can prove that this algorithm correctly outputs the top-k discriminative regions based on the redundancy-aware definition.

In fact, this algorithm generalizes Algorithm 4.1 in two ways. First, in terms of the similarity threshold, if we set $\theta = 1$, the two algorithms will output exactly the same set of top-k results. In terms of pruning power, if we set $\theta = 1$, the two algorithms will verify the same set of regions. The only additional cost of Algorithm 4.2 lies in computing the similarity function $Sim(\cdot, \cdot)$ (Line 7), which can be ignored compared to the overall cost. Also, the generalized algorithm guarantees that `GetRank`() will be called at most once for any region.

To construct the optimal RepCube, extensions of the cost model are needed to accommodate the discriminative semantics. Given a discriminative query $Q(\tau, k, \theta)$ and a position vector $\vec{\phi}$, we have to compute the unpruned set of regions (*i.e.*, all regions where `GetRank`() has been called) and $COST(Q|\vec{\phi})$. Computing the optimal positive vector will be left for future study.

4.5 Relaxing Cells for Space Reduction

In this section, we study techniques to further reduce the storage overhead of the RepCube. The idea here is to merge multiple p-cells with similar aggregate scores and represent them using a single *relaxed cell*. Specifically, instead of materializing the exact scores of a p-cell, we store *score ranges* within a predefined bound. Thus, other p-cells whose exact scores are *covered* by these score ranges can be represented by the relaxed cell.

Definition 13 (ϵ-Relaxed Cell) *Given a region R's p-cell, $PCell = \{f_1, f_2, \ldots, f_m\}$, define its normalized cell as $\{1.0, \frac{f_2}{f_1}, \ldots, \frac{f_m}{f_1}\}$. Given a relaxation parameter $\epsilon \geq 0$, define the corresponding ϵ-relaxed p-cell as $RCell = \{1.0 \pm \frac{\epsilon}{m}, \frac{f_2}{f_1} \pm \frac{\epsilon}{m}, \ldots, \frac{f_m}{f_1} \pm \frac{\epsilon}{m}\}$.*

We elaborate on this definition. The normalization step normalizes aggregate scores that could be at very different scales for subsequent cell merging. This step is important as we observe that different regions may have very similar trends in aggregate score distributions but the absolute values could be quite different. For example, the distribution of the *SUM* of sales in $\{Year = 2009\}$ could be similar to that in $\{Year = 2008 \sim 2009\}$ but differ by a factor 2 in scale. Since in any p-cell f_1 is the largest aggregate score, dividing each cell value by f_1 would make all normalized values to be within range $[0, 1]$. In principle, other normalization methods may also be applied here for the same purpose. Given the relaxation parameter ϵ, each value in an ϵ-relaxed cell would represent a set of ranges of aggregate scores. Specifically, the i-th value of the relaxed cell represents $[f_i - f_1 \times \frac{\epsilon}{m}, f_i + f_1 \times \frac{\epsilon}{m}]$ if f_1 is known. Now, we are ready to introduce a more compact RepCube structure based on a set of ϵ-relaxed cells.

Definition 14 (ϵ-Relaxed RepCube) *An ϵ-relaxed RepCube consists of a collection of r ϵ-relaxed cells $\{RCell^1, RCell^2, \ldots, RCell^r\}$ ($1 \leq r \leq |\mathcal{R}|$), and a surjective mapping function g from each R_i to some relaxed cell, i.e., $g : \{1, 2, \ldots, |\mathcal{R}|\} \rightarrow \{1, 2, \ldots, r\}$. Also the normalization score f_1 is stored for each R_i.*

The ϵ-relaxed RepCube contains no more than r relaxed cells. This means that one or more regions are mapped to a same relaxed cell. We require that these regions' p-cells be covered by the ranges of the relaxed cell they are mapped to. Notice that each region still maintains m values, so the total size of a relaxed cube would be much smaller than the original cube when $r \ll |\mathcal{R}|$.

The query execution algorithm in Table 4.3 need slight modification at the pruning phase (Lines 3–4) to accommodate the relaxed cube. Given R_i, $F(\tau, R_i)$, and a relaxed cell $RCell^{g(i)} = \{1.0 \pm \frac{\epsilon}{m}, \frac{f_2}{f_1} \pm \frac{\epsilon}{m}, \ldots, \frac{f_m}{f_1} \pm \frac{\epsilon}{m}\}$, the computation of $LBRank_i$ (Line 3) now should be computed as $\phi_j + 1$ for j satisfying $f_j - \frac{\epsilon}{m} \times f_1 > F(\tau, R_i) \geq f_{j+1} - \frac{\epsilon}{m} \times f_1$. Similarly, $UBRank_i$ (Line 4) should be $\phi_l - 1$ for l satisfying $f_{l-1} + \frac{\epsilon}{m} \times f_1 \geq F(\tau, R_i) > f_l + \frac{\epsilon}{m} \times f_1$. Note that these bounds guarantee the precision of the query execution algorithm. The pruning power of the relaxed RepCube, on the other hand, will be similar to the original RepCube when the relaxation parameter ϵ is chosen to be very small.

4.5.1 A Greedy Algorithm

Given an original RepCube with $|\mathcal{R}|$ p-cells, we would like to select the smallest subset of their corresponding relaxed cells (*i.e.*, to minimize r) with the constraint that each p-cell must be covered by some relaxed cell. It turns out that this problem is NP-hard with a reduction from the SETCOVER problem: the set of original p-cells are transformed to the *universe of elements* and each potential relaxed cell is transformed to a *set of elements*.

Due to the hardness of the problem, we use a greedy algorithm to iteratively select relaxed p-cells as follows. First, initialize the set of selected relaxed cells as an empty set, and mark all p-cells as "uncovered". Then, add the relaxed cell that is able to cover the largest number of uncovered p-cells into the selected set, and mark those newly covered p-cells as "covered". Repeat the above step until all p-cells are marked as "covered". The corresponding mapping function can be maintained during the above process. Finally the selected relaxed p-cells will be kept in memory or written back to disk.

The parameter ϵ controls the resulting size of the relaxed RepCube. When $\epsilon = 0$, only identical p-cells will be merged. On the other hand, when ϵ is too large, a single relaxed cell suffices to cover all p-cells but is unlikely to provide any pruning power. In effect, a small ϵ less than 0.1 often produces good tradeoff. We manually set it in our experiments and it remains an open problem to automatically determine ϵ.

4.5.2 A Clustering-based Approach

Instead of creating ϵ-relaxed cells, we discuss another clustering-based relaxation approach. Given a set of p-cells, $PCell_1, PCell_2, \ldots, PCell_{|\mathcal{R}|}$, we consider each p-cell as a point in the m-dimensional Euclidean space. For ease of exposition, we let $X = |\mathcal{R}|$ and denote these p-cells as $p_1, p_2, \ldots, p_X$. Now, to reduce the space overhead, we cluster these X m-dimensional points into Y clusters, where Y $(1 \leq Y \leq X)$ is the number of clusters that can be derived from the storage space available. For example, when the storage budget allows 1000 values and $m = 20$, we would generate $Y = 1000/(20 \times 2) = 25$ clusters (we will show that each cluster will store $2m$ values momentarily).

Given these clusters, a clustering-based relaxed RepCube can be generated as the following. Suppose a cluster contains X' points, namely $p'_1 = \{p'_{1,1}, p'_{1,2}, \ldots, p'_{1,m}\}$, $p'_2 = \{p'_{2,1}, p'_{2,2}, \ldots, p'_{2,m}\}$,

$\ldots$, $p'_{X'} = \{p'_{X',1}, p'_{X',2}, \ldots, p'_{X',m}\}$, where $1 \leq X' \leq X$. Then we create a *cluster-based relaxed cell* $RCell = \{[min_i\{p'_{i,1}\}, max_i\{p'_{i,1}\}], [min_i\{p'_{i,2}\}, max_i\{p'_{i,2}\}], \ldots, [min_i\{p'_{i,m}\}, max_i\{p'_{i,m}\}]\}$. In other words, for each of the m dimensions, we use the range between the minimum value and the maximum value of all the X' points at that dimension to represent the relaxed cell. Geometrically, this is equivalent to using the minimum bounding m-dimensional hyper-rectangle to represent all the X' points in the cluster. Clearly, when the clustering quality is good, we have that the hyper-rectangle is "tight", and thus the relaxed cells can provide good pruning power for online query execution. On the other hand, when the clusters are not tight, the pruning power of the relaxed cells will not perform well.

To yield clusters with good quality, a clustering objective must be defined. Intuitively, we require the hyper-rectangles corresponding to the clusters to be as small as possible. Thus, we can formulate the following clustering problem: given X m-dimensional points and a non-negative integer Y, generate a Y-clustering of these points such that the sum of the diameters of these clusters is minimized. Here, the diameter of a cluster is the maximum Euclidean distance between any pair of points in that cluster. It turns out that it is difficult to efficiently obtain an optimal solution of the clustering problem; and several approximation algorithms have been studied [26, 15, 31]. For large data sets, one may simply use a hierarchical clustering or the k-means algorithm to process the data points efficiently.

In comparison to the ϵ-relaxed RepCube, the advantage of such a clustering-based relaxation approach is that users do not need to specify the ϵ threshold. Instead, the number of clusters can be naturally derived from the storage budget given m. Also, the hyper-rectangle corresponding to each cluster will tightly bound all points in that cluster so that no cluster can be further compressed. On the other hand, the quality of such clustering can no longer be guaranteed by a fixed constant because some hyper-rectangle might be quite large in the presence of outlier points. A hybrid approach that combines the advantage of the two relaxation approaches may be developed, which is beyond the scope of this study.

4.5.3 Incremental Updates

We briefly discuss the incremental updating issue for the relaxed cube structure. First, when the aggregate scores of a region is updated, one of the following 3 cases is considered. (1) If these aggregate scores are changed within the ϵ range of the region's relaxed cell, the relaxed cell would remain unchanged. (2) If the change is slightly larger than ϵ, we can locally adjust ϵ for its relaxed cell. (3) Otherwise, a new relaxed cell need to be created to accommodate the change unless the relaxed cell only covers a single p-cell. Similarly, when a new region is created, we can map it to some existing relaxed cell if its p-cell can be covered. Otherwise we have to create a new relaxed cell. It is expected that the relaxed cells are relatively stable in response to updates. A comprehensive study of incremental maintenance is left for future work and we do not evaluate it in this chapter.

4.6 Experiments

We conduct case study on the DBLP data set and comprehensive experiments on the standard TPC-H benchmark. Our goal is to: (1) demonstrate the top-k results through a case study, (2) show that our proposed methods can significantly outperform a baseline solution in terms of *query execution time*, and (3) verify that the *storage space* used by our methods is very small.

All our experiments were done on a machine with a 2.5GHz duo-core CPU, 4GB of RAM, and 250GB hard disk. The OS is Windows XP Pro SP3 and all source code was written and compiled in Microsoft Visual C# 2008.

4.6.1 A Case Study on DBLP

We constructed a fact table from the DBLP data set using *Conference* as the categorical dimension, *Year* as the continuous ranged dimension, *Author* as the object dimension, and *Paper Count* as the measure dimension. The table contains about 1.8 million base tuples and $450K$ authors.

For the query author *Bruce Lindsay*, we found his global rank to be 5112th. However, the top-3 regions (except the global region) based on *PRank* are $\{VLDB, 1990 \sim 1991\}$, $\{ICDE, 1993 \sim 1993\}$, and $\{SIGMOD, 1998 \sim 2002\}$, where he is ranked (5th, top-2.1%), (4th, top-2.2%), and (4th, top-2.8%), respectively. Not surprisingly, the results are quite meaningful.

On the other hand, his *(Rank, PRank)* in the promotion region $\{SIGMOD, 1998 \sim 2002\}$'s five child regions, namely $\{SIGMOD, 1998 \sim 1998\}$ through $\{SIGMOD, 2002 \sim 2002\}$, are only (21st, top-9.3%), (36th, top-14.6%), (28th, top-11.8%), (29th, top-12.6%), and (37th, top-16%), respectively. These results indicate that it is indeed interesting to discover the "right" region for promotion. We leave a systematic evaluation of various other semantics to our future work.

4.6.2 Evaluation on TPCH

For efficiency evaluation, we chose the TPC-H benchmark[1] to generate large decision support data. The *default fact table* was generated as follows. We ran the *dbgen* executable with default parameters to generate a set of data files and extracted the *lineitem.tbl* file containing 6,001,215 base tuples. We set *l_linenumber* (cardinality=7) as the categorical dimension, *l_quantity* (50) and *l_linestatus* (2) as the range dimensions, *l_suppkey* (10000) as the object dimension (*i.e.*, $n = 10000$), and *l_extendedprice* (real numbers) as the measure dimension. Thus, for this default fact table there are totally 30600 regions. This table is stored in Microsoft SQL Server 2008.

Algorithm implementation: We implemented the following 5 methods.

- (Empty) On-the-fly query execution without any auxiliary materialization as a baseline for online query execution time;
- (Full) Precomputing aggregate scores for all objects in all regions as a baseline for storage overhead;
- (Uniform) The uniform RepCube approach;
- (Opt) The optimal RepCube approach; and
- (Relax) The relaxed RepCube approach.

All of these 5 methods rely on 2 interface primitives `GetAgg`() and `GetRank`(). The former primitive was implemented by ourselves, while the latter was implemented as a query in SQL Server. To speedup query processing, a clustered index was built on the object dimension and

[1] http://www.tpc.org/tpch/

multi-key non-clustered indices were built on categorical and range dimensions. All materialization files were stored as plain text files.

To formulate a cost model and derive the optimal position vector for Opt, we set Ω, the constant cost of the pruning phase, to 0, and let $COST(s_i)$ be 1 for any region R_{s_i} (see Section 4.4 for the definitions of Ω and $COST(s_i)$).

To produce the set of top-k regions, *SUM* was used as the aggregation function F and we consider the k regions with the largest *percentile rank* as the top-k results.

Performance measure: We focus on *runtime* (*i.e.*, average online query processing time per query, in terms of *seconds*) and *size* (*i.e.*, offline storage space an algorithm is used, in terms of *number of values* stored) as the main performance metrics of these algorithms. We do not count the time for loading the materialized data.

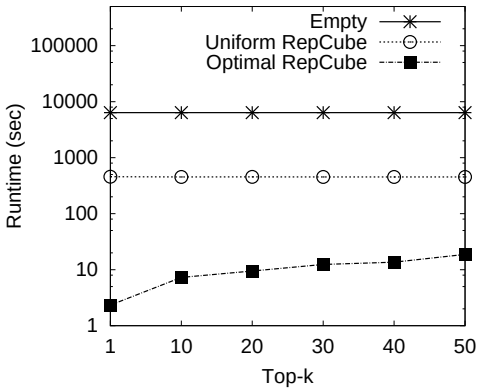
4.6.3 Online Query Execution Time vs. Top-k

Now we compare the runtime of Empty, Uniform, and Opt by varying the query parameter k. The performance results for Relax will be reported shortly.

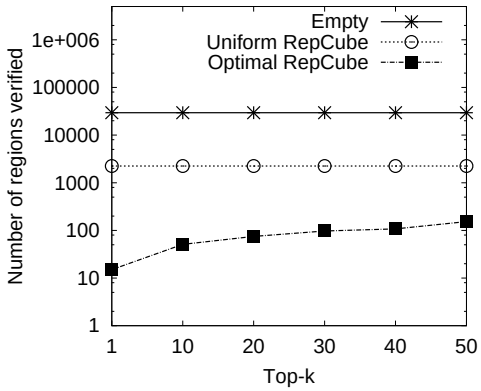
We set $m = 10$ for both Uniform and Opt such that the resulting size of Uniform is 367,201 values and that of Opt is 367,210 values. To compute the position vector for Opt based on the cost model, we generated a *default workload* consisting of 200 promotion queries $Q(\tau, k)$, where for each query τ was uniformly randomly generated and k was uniformly randomly distributed over $[1, 160]$.

A set of 5 *random test queries* was generated as follows: 5 objects were uniformly randomly generated, and k was varied from 1 to 50. Figure 4.1(a) displays the average runtime (in log-scale) of Empty, Uniform, and Opt on these 5 test queries. We can see that the baseline solution Empty is over an order of magnitude slower than Uniform, the basic RepCube strategy, while Empty is over 3 orders of magnitude slower than Opt at $k = 1$, and > 300 times slower than Opt at $k = 50$. Also, the performance of Empty does not change with respect to top-k because it does not involve any pruning. This test clearly shows that computing the region-based promotion query from scratch can be prohibitively expensive. In our subsequent experiments we will not evaluate Empty any more due to its apparent low efficiency.

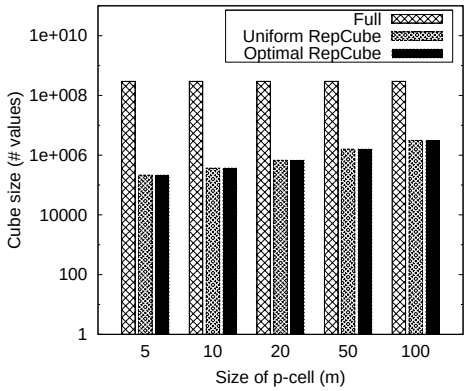
Observe that Uniform it is 190 times slower than Opt at $k = 1$ and 24 times slower at $k = 50$.



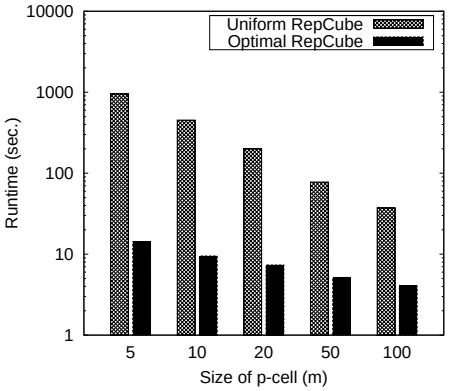
(a) Runtime vs. top-k.



(b) Number of verified regions vs. top-k.



(c) Storage overhead vs. the size of p-cell



(d) Runtime vs. the size of p-cell

Figure 4.1: Comparison with baseline solutions on the default TPCH data set.

The Uniform approach, with some precomputed information, is able to significantly outperform Empty, but it turns out to be quite insensitive to k as well, because it is not able to leverage the fact that the object of interest is often highly ranked in the top-k promotion regions; in other words, the pruning power of Uniform would be similar no matter the object is highly ranked in the top-k regions or not. On the contrary, Opt offers much better pruning power as it is able to precompute sample aggregate scores in an adaptive way. As a result, it is more sensitive to k and more efficient when k is smaller.

To accurately explain the gap of runtime between different methods, in Figure 4.1(b) we plot the average number of verified regions (*i.e.*, the number of unpruned regions as shown in Table 4.3, Line 7) with respect to k in the same test. As can be seen, this figure matches Figure 4.1(a) well. This validates our claim that the query execution time is dominated by the cost of `GetRank`(). Indeed, Empty need on average about 30000 calls of `GetRank`() (note that there are some regions where the object of interest does not appear so `GetRank`() does not have to be called for them), while Uniform and Opt need no more than 2300 and 160 calls for any k, respectively.

4.6.4 Storage Overhead vs. P-Cell Size

Now we compare the storage overhead of Full, yet another baseline strategy, with Uniform and Opt. We vary m, the size of p-cell from 5 to 100 (*i.e.*, 5 to 100 aggregate scores are sampled for each region) and show the resulting storage space required by each method in Figure 4.1(c). Fully precomputing aggregate scores for all objects in all regions requires about $3 * 10^8$ values. Suppose each aggregate score uses 8 bytes to store, Full would consume 2.2GB of disk space, which is much larger than the size of the input data set. This tells us that Full may not be a practical solution for large applications. In contrast, Uniform and Opt store no more than $220K$ values (1/1380 of Full's size) at $m = 5$; $370K$ values (1/805 of Full's size) at $m = 10$; and $3.2M$ values (1/94 of Full's size) at $m = 100$, which significantly alleviate the problem of expensive storage overhead. Note that the difference of storage overhead between Uniform and Opt is very small ($< 10^{-4}$ of the total cube size). In principle, the resulting size of Opt is linearly related to m, so users are able to conveniently specify m to control the size and obtain their desired performance.

Figure 4.1(d) further displays how the p-cell's size would affect the online performance of our

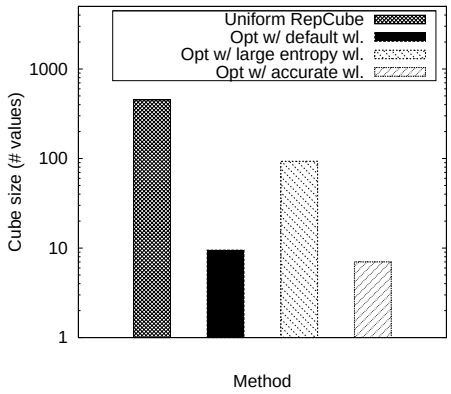


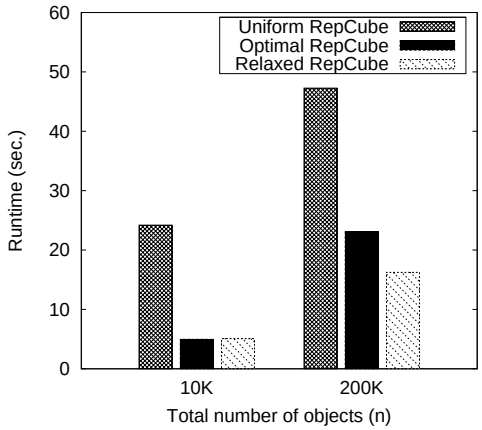
Figure 4.2: Query execution time vs. query distribution.

proposed methods. For each m, we used exactly the same set of 5 test queries and fix k to 20. We also used the same default workload to generate Opt's position vector as we did for the previous test. The average query execution time is reported for both Uniform and Opt. We can see that when m increases, the efficiency of both Uniform and Opt becomes higher. This is expected as increasing the p-cell size would help derive tighter upper- and lower- bounds for any object in any region, thereby leading to more pruned regions.

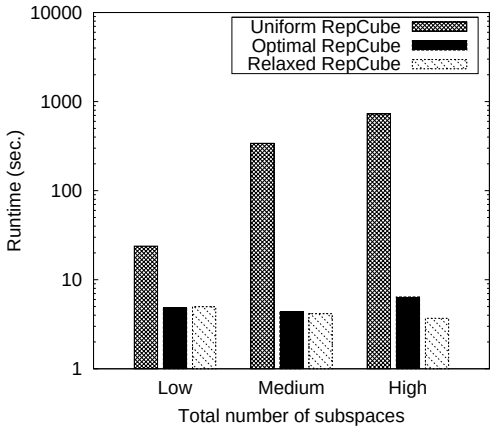
The relation between Uniform and Opt as shown in Figure 4.1(d) is also interesting. When $m = 5$, Opt is 66 times faster than Uniform. As m is increasing, the gap between the two approaches actually become smaller. For example, the speedup ratio is 15 at $m = 50$ and 9.1 at $m = 100$. Indeed, too large m might lead to a convergence of Opt to Uniform; an extreme case is that when $m = n$, the performance of Opt and Uniform would be identical due to a same position vector. Nevertheless, this result validates our idea that Opt performs much better than Uniform when m is reasonably small (*e.g.*, $\frac{m}{n} < 1\%$), which is desirable for large data sets.

4.6.5 Performance of the Optimal RepCube over Different Query Distributions

In our previous tests we computed Opt's position vector $\vec{\phi}$ using the default workload, *i.e.*, we assumed that in the query distribution object IDs were drawn uniformly and k uniformly randomly from 1 through 160. Because in different applications, such query distributions can be vastly different, in this subsection we test the performance of Opt using different workloads. Besides the default workload used earlier, we generated two other workloads:



(a) Query execution time vs. number of objects (n).



(b) Query execution time vs. number of regions ($|\mathcal{R}|$).

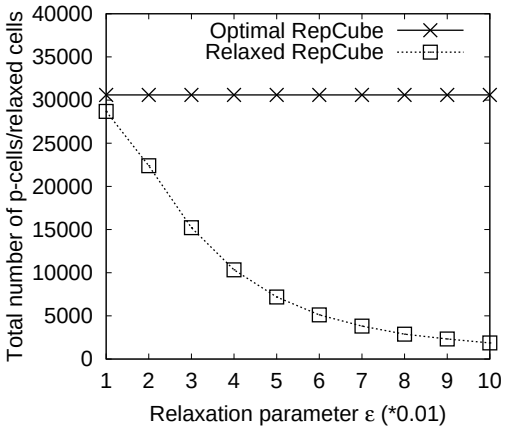
Figure 4.3: Performance results with different data characteristics.

- *Large-entropy workload:* Consists of queries with τ uniformly randomly sampled from the set of all objects and k uniformly distributed on 1 through 10000. In other words, this workload assumes the promotion query has a large entropy (*i.e.*, randomness).

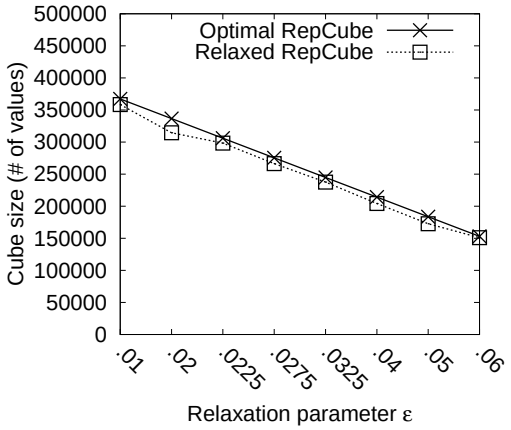
- *Accurate workload:* Given the 5 test queries used earlier, we generated a superset of them as an accurate workload (totally 10 queries). Therefore, Opt would be specifically optimized for the test queries using this workload.

We ran Opt against the 3 workloads and obtained 3 position vectors. Figure 4.2 plots the average query execution time of Opt vs. these workloads over the 5 test queries; we also plot the result of Uniform for comparison. Interestingly, we observe that Opt outperforms Uniform in all 3 cases. In particular, Opt based on large-entropy workload runs 5 times faster than Uniform, whereas Opt based on the accurate workload does 60 times faster. These results match our intuition, because the large-entropy workload can be considered as "adversarial" since the query distribution used by the cost model largely deviates from test queries, while the accurate workload would help maximize the pruning power of the optimal RepCube generated for those test queries. The gap between Uniform and Opt with the large-entropy workload, however, is unexpected.

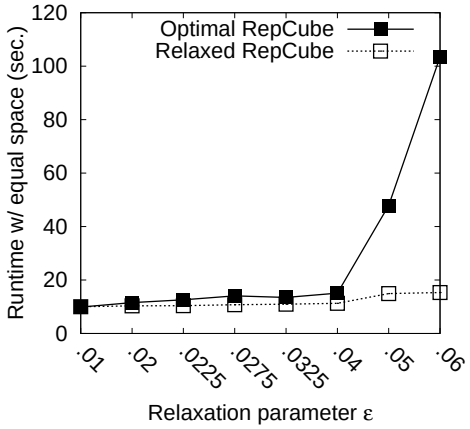
To clearly explain the performance gap, Table 4.6 shows the position vectors used by each method. Uniform uses 10 evenly spaced positions. For Opt, however, the position vectors have smaller values; in fact, even the uniform workload "prefers" smaller position values strictly based



(a) Num. of relaxed cells generated by the greedy algorithm.



(b) Size of Opt (by varying m) and Relax (by varying ϵ).



(c) Query execution time vs. storage overhead

Figure 4.4: Performance results on the relaxed RepCube.

on the cost model. Therefore, these results empirically proved that choosing an evenly spaced position vector (*i.e.*, uniform quantization) cannot produce desirable pruning power for promotion query.

4.6.6 Performance of the Relaxed RepCube

Now we evaluate Relax. Recall that Relax's parameters m and ϵ dictates the resulting size of materialization. Based on the optimal RepCube ($m = 10$) for the default fact table discussed earlier, we varied ϵ and ran the greedy relaxed cell selection algorithm. Figure 4.4(a) depicts the resulting size of Relax for ϵ ranging from 0.01 to 0.1 on an increment of 0.01. While Opt must store 30600 p-cells constantly, Relax need fewer and fewer relaxed cells as ϵ increases. For example, when

RepCube method	Position vector $\vec{\phi}$
Uniform	1, 1001, 2001, 3001, 4001, 5001, 6001, 7001, 8001, 9001
Opt + Default	6, 18, 33, 48, 89, 142, 234, 357, 593, 1084
Opt + Large entropy	214, 659, 1009, 1621, 1869, 2465, 3326, 3726, 4416, 5448
Opt + Accurate	4, 7, 23, 26, 76, 111, 132, 187, 283, 328

Table 4.6: Position vectors used by different RepCube methods.

$\epsilon = 0.01$, 28695 relaxed cells can cover all p-cells, and when $\epsilon = 0.1$, only 1865 cells would suffice, where each relaxed cell covers an average of 16.4 p-cells. Hence the result confirms that similar p-cells and can be merged effectively.

Let us turn to a comparison between Relax and Opt. Since it is unfair to compare their runtime using different amounts of storage space, the methodology adopted here is to first generate Relax and Opt with similar size, elaborated as follows. First, we generated a set of 8 optimal RepCubes by varying the p-cell size m from 10 down to 3. The resulting sizes of Opt ranges from 367,210 to 153,003. Second, to generate a relaxed cube with comparable size to each of the 8 optimal RepCubes, we manually tried different ϵ parameters and ran the greedy algorithm on Opt with $m = 10$. Finally we chose ϵ to be 0.01, 0.02, 0.0225, 0.0275, 0.0325, 0.04, 0.05, and 0.06, respectively, and Figure 4.4(b) displays the size of both Opt and Relax in the 8 cases. For instance, when $m = 8$ for Opt, we set ϵ to 0.0225, and then Opt's size is slightly above $300K$ and Relax's size is slightly less than $300K$. We guarantee that the size of Relax be no larger than Opt in all cases.

For each matched pair of Relax and Opt, we ran the 5 test queries and reported the average query execution time in Figure 4.4(c). The figure shows that Relax beats Opt in all but the first case. Relax is considerably more efficient than Opt in the last two cases. For the second to last case, Relax with $\epsilon = 0.05$ is 3.2 times faster than Opt with $m = 4$; whereas for the last case, Relax with $\epsilon = 0.06$ is 6.7 times faster than Opt with $m = 3$. Even in the first case the performance gap can be neglected. Hence, our conclusion is that Relax indeed gives the best tradeoff between storage space and query execution time, since it is faster than Opt yet using less space.

4.6.7 Varying Data Characteristics

To compare the performance of the proposed methods on different data characteristics, we first generated a new fact table using the $6M$-tuple *lineitem* table. We fixed the categorical dimension

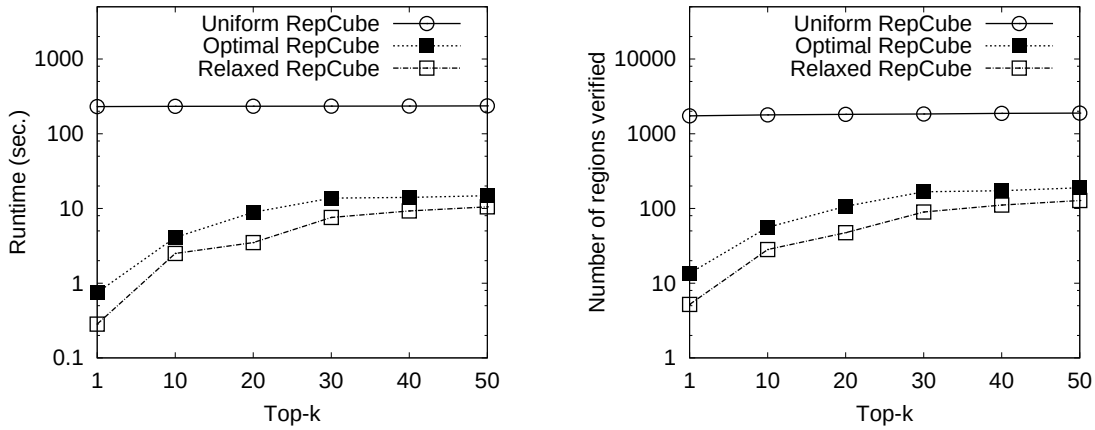
to *l_linenumber* (7) but changed range dimensions to *l_discount* (11) and *l_linestatus* (2). We fixed the measure dimension to *l_extendedprice*, whereas the object dimension was changed to *l_partkey* ($200K$), indicating that the number of object is $200K$. The workload was the default one and the test queries were the same before.

The new data with $n = 200K$ contains a total number of 1584 regions. Full would consume more than $187M$ values, or $1.4G$ of disk space equivalently. When setting m to 10, Uniform and Opt have about the size of $19K$ values, about $\frac{1}{9850}$ of Full's size. We can see that the RepCube approaches achieve a better storage saving for larger number of objects as expected. For Relax, we first generated an optimal RepCube with $m = 18$ and then set ϵ to 0.059 to produce a relaxed RepCube using $< 19K$ values (701 relaxed cells generated).

As shown in Figure 4.3(a), Opt outperforms Uniform by 2 times. Since the total number of regions is smaller than in the previous test cases, the speedup ratio is not as large as in the previous tests; in fact, this ratio would increase with respect to $|\mathcal{R}|$ as will be shown shortly. Relax is in turn 1.4 times faster than Opt, thereby again beating Opt in both storage overhead and runtime. It also turns out that for $n = 10K$ (*i.e.*, using *l_suppkey* ($10K$) as the object dimension while keeping other dimensions fixed), the performance of Opt and Relax is similar due to fewer regions.

Let us turn to the total number of regions. In previous tests we have synthesized two fact tables with 1584 regions (denoted by "*LOW*" hereafter) and 30600 regions (denoted by "*MEDIUM*" hereafter), respectively. In addition to *LOW* and *MEDIUM*, we produced another *HIGH* data set from the *lineitem* table by setting *l_quantity* (50) and *l_returnflag* (3) as range dimensions while keeping other dimensions fixed ($10K$ objects). *HIGH* contains 61156 regions and the Full approach would generate $> 580M$ values ($4.5G$ disk space). For *HIGH*, we again set m to 10 and generated Uniform and Opt with considerably smaller space overhead (*i.e.*, $< 734K$ values), while repeated the previous approach for Relax (*i.e.*, $< 730K$ values and each relaxed cell on average covers 2.16 p-cells). Figure 4.3(b) shows the performance comparison of Uniform, Opt, and Relax on *LOW*, *MEDIUM*, and *HIGH*, respectively. We can see that (1) Opt becomes increasingly faster than Uniform, *i.e.*, 4.9 times faster on *LOW* and 113.6 times on *HIGH*; and (2) Relax again shows its scalability, *i.e.*, using less space than Opt yet being 1.7 times faster than it on *HIGH*.

Based on the experimental results displayed in Figure 4.3, our conclusions are: (1) Both Opt



(a) Runtime vs. top-k on *AVG*.

(b) Number of verified regions vs. top-k.

Figure 4.5: Performance results on aggregate function AVG.

and Relax perform consistently and significantly faster than the basic RepCube implementation (Uniform) during online processing; and (2) Relax, although using less space, is more efficient than Opt on large data sets, demonstrating its scalable tradeoff in terms of the number of objects as well as regions.

4.6.8 Performance on Aggregate Function AVG

Our final test case in Figure 4.5 compares Uniform, Opt, and Relax based on another aggregate function *AVG*. That is, in each region, objects are ordered descendingly according to their average measure dimension values. The *MEDIUM* fact table is used here. The generation of Uniform and Opt remains unchanged with $m = 10$. For Relax, we first generated an optimal RepCube using $m = 18$. Then, ϵ was set to 0.027 such that we ensure Relax's size be smaller than Opt; this is a notable difference between *AVG* and *SUM* in that here the p-cells can be merged more easily using a smaller value of ϵ. This indicates that each relaxed cell represents a "tighter" range than for the *SUM* aggregate function.

Figure 4.5(a) reports the runtime of the methods when varying k from 1 to 50. We can see that Uniform does not have satisfactory performance, while Relax is consistently about 2 times faster than Opt. Figure 4.5(b) further confirms that the verification step dominates the query execution cost, which is invariant to the aggregate function. The results obtained here thus prove that the efficiency gain of Opt and Relax are not restricted to some particular monotone measures.

4.7 Summary

This chapter studied a novel class of decision support queries called the top-k (discriminative) region-based promotion query. A region-based promotion cube framework was developed. We showed that a uniform materialization approach is indeed not the best; instead, an adaptive approach was developed based on a solid theoretical analysis to produce the provably optimal structure. In addition, a compact relaxed cube structure was studied to further optimize storage overhead. Comprehensive experiments on both real and synthetic data sets verified both the effectiveness and efficiency our proposed techniques.

Chapter 5

Supporting Rank-Driven Top-K Object Queries

5.1 Introduction

In business intelligence applications, it is crucial to support online analytical processing (OLAP) and explorative mining tasks. Typically, the source data underlying such tasks comprises a large number of objects such as products or business entities. These objects are often associated with multiple feature dimensions and thus can be consolidated into a multidimensional fact table. Unlike the previous chapters, where objects are drawn from a single-typed, homogeneous collection, here we assume that objects can form multidimensional relationships.

Example 14 (Multidimensional Object Data) *Table 5.1 illustrates an example automobile data warehouse containing a collection of products. Each row in this database refers to a basic product described by multiple dimensions. For example,* (Ford, Fusion, 2009, FWD) *refers to a 2009 car model manufactured by Ford with the "Drive" dimension equal to "FWD". Also, each of these basic products is associated with a measure dimension,* Rating*, that records the customer rating for each basic product. Based on such multidimensional organization, basic products sharing the same dimension values can be grouped to form products in higher-level spaces, whose measure*

Make	Model	Year	Drive	Rating
Ford	Fusion	2009	FWD	3.2
Ford	Fusion	2010	AWD	4.0
Ford	Fusion	2010	FWD	4.1
Ford	Explorer	2009	AWD	3.4
Ford	Explorer	2010	AWD	4.2
GMC	Acadia	2009	AWD	4.0
GMC	Acadia	2010	FWD	4.9
GMC	Sierra	2009	AWD	3.7

Table 5.1: Example multidimensional view of objects.

can be aggregated correspondingly. In Table 5.1, there are totally 8 distinct product spaces {Make}, {Make, Year}, {Make, Drive}, {Make, Year, Drive}, {Model}, {Model, Year}, {Model, Drive}, *and* {Model, Year, Drive}. *A high-level product, say,* (Fusion) *has 8 sub-products,* (Fusion) *itself,* (Fusion, 2009), (Fusion, 2010), (Fusion, FWD), (Fusion, AWD), (Fusion, 2009, FWD), (Fusion, 2010, AWD), *and* (Fusion, 2010, AWD). *The aggregate measure of* (Fusion) *can be computed using "average" as* $(3.2 + 4.0 + 4.1)/3 \approx 3.77$.

In real-world data sets, a large number of products coupled with their interrelated relationships formed over the multidimensional space makes it very difficult for analysts to gain insights through simple OLAP queries. While existing systems empower users to make decisions through various aggregation functionalities, in this chapter we tackle the promotion analysis problem from a different perspective and study a new problem called the *Rank-Driven Top-k Object Search Query* (kOSearch), which complements the previous chapters. Intuitively, given a query product (object) p of interest, we would like to find the k most highly ranked sub-products (descendent objects) of p. The formulation of this type of queries is illustrated in the following.

Example 15 (Rank-Driven Top-k Object Search) *Continue from Table 5.1. In each of the 8 product spaces, we can obtain a product ranking by sorting all products in it descendingly in terms of their aggregated rating. For instance, the product* (Fusion)*'s rank is 3rd, since in its object space* {Model} *we have the product ranking* (Acadia:4.45) > (Explorer:3.8) > (Fusion:3.77) > (Sierra:3.7) *based on the average rating. This way, we can obtain the rank of all products. Now suppose that a query product* $p =$ (Fusion) *is given, then p's top-3 most highly ranked sub-products are* (Fusion, AWD: 2nd), (Fusion: 3rd), *and* (Fusion, 2010, FWD: 3rd) *because no other sub-product of p is ranked better than 3rd.*

Note that in the kOSearch problem, *rank* is computed using multidimensional aggregates, which is subsequently used as the measure to judge the interestingness of a sub-product. We will formalize this model in Section 5.2. This problem often arises in real applications, and the results of kOSearch can serve the promotional purposes for the query. We present several scenarios:

1. Discover the "best" sub-products and features of a given product of interest. This enables data analysts to understand the strengths of their products for further decision making.

In a more general sense, it helps detect anomaly or extreme aggregates in different spaces. The kOSearch query cannot be replaced by a traditional iceberg query [30] that returns objects passing a user-specified hard threshold, because (1) the number of answers cannot be controlled by the iceberg query, and (2) one threshold cannot fit different product spaces, where the aggregates are at different scales and not comparable;

2. Facilitate explorative and progressive analysis. Since rank is not a monotone measure, users who are seeking interesting, highly ranked products must drill down and roll up in the product space level-by-level in a trial-and-error fashion. This problem can be addressed by the kOSearch query. For example, a data analyst may issue "Dell" as a query and first search for their most popular/successful products across the top-3 level spaces, where she can find highly ranked products such as "(*Home, Gaming, Desktop*)". Then, "(*Home*, *Gaming, Desktop*)" is subsequently submitted by her as a new query so as to drill down to explore the highly ranked features of home gaming desktops, such as "(*19inch, LCD*, *4GB Memory*)".

Based on the problem formulation, several variants are possible: (1) to avoid drilling too deep in the product space (*i.e.*, a combination of many dimension values is unlikely to make much sense); (2) to avoid overlaps of top-k results (*i.e.*, to prevent that some top-k answers from being too similar with each other); and, (3) to weigh product spaces or ranks differently. Nonetheless, the techniques developed in this chapter are applicable to all these variants with proper extensions (the discussion is deferred to Section 5.6).

The technical challenges for answering kOSearch queries lie in that a query object (product) p often induces a large number of sub-products, which are likely to occur in $\Theta(2^n)$ product spaces given a total number of n dimensions. Moreover, computing the rank of each sub-product from scratch can be prohibitively expensive since aggregating each product space incurs in the worst case a full scan of the whole data set. On the other hand, fully materializing all possible object rankings can render storage cost infeasible, especially in the presence of high dimensionality. Also, previous exact or probabilistic top-k query processing techniques are not applicable since none of them can compute objects' rank efficiently.

To address these challenges, we study two categories of indexing techniques, namely *exact* and *approximate* techniques. We first study two simple exact strategies, namely the *horizontal score*

index and the *vertical score index*, whose goal is to support exact top-k computation and yield the best tradeoff between efficiency and storage overhead. These strategies will be considered as the baseline. Then, we further develop a *probabilistic score index* strategy for approximating top-k answers. By leveraging piecewise regression modeling and residual error computation, this strategy is able to provide probabilistic guarantee to top-k answers' precision according to any user-specified confidence threshold. Also, a lightweight random sampling method is developed to compute such confidence efficiently with theoretical guarantee. We show that, while sacrificing very little or virtually no result quality, the probabilistic strategy can consistently achieve significantly better tradeoff between efficiency and storage overhead in comparison to the baseline strategies (*i.e.*, 1 to 2 orders of magnitude more efficient while having only 1/100 of the storage overhead). In summary, this chapter makes the following contributions:

- Propose and formulate the *Rank-Driven Top-k Object Search* (kOSearch) problem, which, to the best of our knowledge, has not been studied before (Section 5.2);
- Discuss exact query processing strategies (Section 5.3);
- Develop the *probabilistic score index*, a general, principled framework for approximate query processing with probabilistic guarantee (Section 5.4);
- Conduct case study and performance evaluation on real and synthetic data sets to verify the effectiveness and efficiency of our proposed model and algorithms (Section 5.5).

In addition, Section 5.6 discusses several extensions of the techniques developed in this chapter, and finally Section 5.7 presents a conclusion.

5.2 Problem Formulation

Consider a fact table with d dimensions, $D_1, D_2, \ldots, D_d$, and a measure dimension *Measure*. Each dimension $D_{d'}$ $(1 \leq d' \leq d)$ belongs to a finite, categorical domain and let the domain of *Measure* be real values. These d dimensions induce a lattice $\mathbb{S}$ consisting of 2^d *object spaces* . For example, given $d = 3$ dimensions, A, B, and C, Figure 5.1 depicts the corresponding lattice of the object spaces. In particular, the *base* object space, denoted by $\{ABC\}$, contains the objects at the lowest

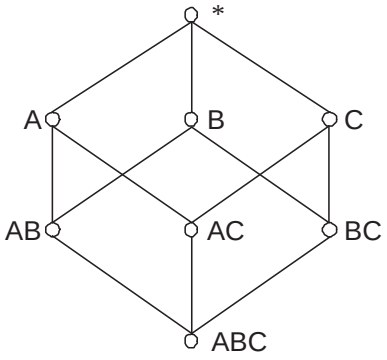


Figure 5.1: Object space lattice.

granularity. The *apex* object space, denoted by $\{*\}$, on the other hand, is at the highest level, representing the grouping of all base tuples.

In real applications, users may not be interested in all of the objects spaces. They may choose to ignore some too small or too large spaces, or impose weights on different spaces. For clarity of presentation, we assume users are interested in L (a given number between 1 and 2^d) of all the object spaces, denoted by $S_1, S_2, \ldots, S_L$ ($S_l \in \mathbb{S}$).

Each object space S_l ($1 \leq l \leq L$) contains a collection of $|S_l|$ distinct *objects*. An object is a base objects if it belongs to the base object space, or an aggregate object otherwise. Given an arbitrary object (base or aggregate) p, we use a function $S(p)$ to indicate the object space where p has membership (*i.e.*, p is contained in $S(p)$). For instance, in Table 5.1, each row corresponds to a distinct base object. An example aggregate object $p = (\textit{Ford}, \textit{2010})$ belongs to the 2-dimensional object space $\{\textit{Make}, \textit{Year}\}$ (*i.e.*, $S(p) = \{\textit{Make}, \textit{Year}\}$).

Definition 15 (Descendant Object) *Given two objects p and q, we say that q is a* descendant object *of p if and only if $S(p) \subseteq S(q)$ and the set of q's corresponding base objects is a subset of p's. We use $C(p)$ to represent the complete set of p's descendant objects.*

The relationship between an object and its descendent objects models the relationship between a product and its sub-products or features discussed earlier. Trivially, any object p is a descendant of itself. To illustrate the concepts, Figure 5.2 displays three example object spaces, $\{A\}$, $\{AB\}$, and $\{AC\}$, along with the collection of objects each contains. For instance, we have $S(a_1) = \{A\}$ and $C(a_1) = \{(a_1), (a_1b_5), (a_1b_2), (a_1c_1)\}$ for an object (a_1). We omit other object spaces and

p (object)	$M(p)$ (score)
a_3	2.1
a_6	1.9
a_1	1.6
a_2	1.5
a_4	0.8
a_5	0.7
...	...

sorted list of $\{A\}$

p (object)	$M(p)$ (score)
a_2b_3	2.5
a_5b_4	1.3
a_2b_3	1.2
a_1b_5	1.0
a_4b_6	0.5
a_1b_2	0.2
...	...

sorted list of $\{AB\}$

p (object)	$M(p)$ (score)
a_5c_3	1.8
a_6c_2	1.6
a_2c_2	1.6
a_3c_3	1.0
a_1c_1	0.6
a_2c_3	0.5
...	...

sorted list of $\{AC\}$

... other spaces omitted ...

Figure 5.2: Example object spaces and sorted lists of scores.

objects not shown in the figure.

Given an aggregate function such as *AVG* (average), *SUM*, or *VAR* (variance), we can compute the *score* of any object by aggregating the scores of all its descendent base objects. In this chapter, we assume that an aggregate function M is given, and denote by $M(p)$ the score of any object p. Note that we do not require M to be a monotone or algebraic function. All techniques developed in this chapter can be applied toward any *ad-hoc* aggregate functions.

Because our goal is to discover objects which are highly ranked in their spaces, we define the *rank* measure of an object p as its relative rank of score among all scores in $S(p)$.

Definition 16 (Object Rank) *Given any object p, we have*

$$Rank(p) = \frac{\big|q|q \in S(p) \wedge M(q) > M(p)\big| + 1}{|S(p)|}.$$

Without loss of generality, assume in this definition that objects are ordered descendingly in terms of scores. Also, the rank of p is normalized by the total number of distinct objects of the space p belongs to, so we have $Rank(p) \in (\frac{1}{|S(p)|}, 1]$.

For ease of exposition, we denote by $\{m_1^l, m_2^l, \ldots, m_{|S_l|}^l\}$ the *sorted list of scores* in each object

space S_l. Throughout the chapter, when the object space S_l is clear in the context, we will ignore the superscript and put $\{m_1, m_2, \ldots, m_{|S_l|}\}$ for short.

Continue from the example in Figure 5.2. The sorted list of scores is shown for each object space (right column). We can see $M(a_1) = 1.6$. Supposing $\{A\}$ contains 100 objects in total (*i.e.*, $|\{A\}| = 100$), we will have $\mathit{Rank}(a_1) = 3/100 = 0.03$.

Now we formulate the *Rank-Driven Top-k Object Search* (kOSearch) problem as follows.

Definition 17 (Rank-driven Top-k Object Search Query) *Given a query object p, our goal is to return $C^k(p)$ ($\subseteq C(p)$), the ordered list of the top-k descendent objects of p that are the most highly ranked; that is, $\forall q \in C^k(p) \wedge \forall q' \in C(p) \setminus C^k(p) \Rightarrow \mathit{Rank}(q) \leq \mathit{Rank}(q')$.*

Ties are broken arbitrarily. We illustrate this query model using the following example.

Example 16 *Following from the example in Figure 5.2, let us consider a query object (a_1) and $k = 2$. Suppose $\{A\}$, $\{AB\}$, and $\{AC\}$ contain 100, 200, and 200 objects, respectively. For each of (a_1)'s descendent objects, we can compute $\mathit{Rank}(a_1) = 3/100 = 0.03$, $\mathit{Rank}(a_1b_5) = 4/200 = 0.02$, $\mathit{Rank}(a_1b_2) = 6/200 = 0.03$, and $\mathit{Rank}(a_1c_1) = 5/200 = 0.025$. Therefore, the top-2 objects would be (a_1b_5) and (a_1c_1).*

While the query model in Definition 17 uses a simple relative object ranking to quantify the "top-k interesting" objects, in practice there can be several other variants to model more complex semantics. First, *absolute rank* instead of relative rank may be used. In Example 16, the top-2 objects would be (a_1) and (a_1b_5) according to absolute rank. Second, object spaces may be weighed differently by different users. For example, some users may treat $S_1, S_2, \ldots, S_L$ equally, while others think some particular spaces are more important. In general, a set of non-negative weights, $w_1, w_2, \ldots, w_L$ can be specified at query time. Similarly, different rank positions can have different weights as well. Third, sometimes it may not be desirable for the top-k results to contain similar objects. For instance, two objects (*Ford*, 2009) and (*Ford*, 2010) might be considered redundant. In Section 5.6, we will show that our techniques can be straightforwardly extended to address the first two issues, and an extension to the third issue will be discussed.

5.3 Exact Algorithms

We first study algorithms for exact query processing. We first explore the tradeoff between two naive strategies, namely the no materialization strategy (Section 5.3.1) and the full materialization strategy (Section 5.3.2), which lay the foundation for the horizontal and vertical approaches (Sections 5.3.3 and 5.3.4). The general query execution algorithm is then presented in Section 5.3.5.

5.3.1 No Materialization

Without using any auxiliary data structures, a query p can be executed on-the-fly in two steps. First, for each object space S_l $(1 \leq l \leq L)$, we compute its sorted list of objects and the corresponding scores using a group-by query. Second, the rank of all descendent objects of p can be computed given all the sorted lists, and the top-k descendent objects with the highest ranks are returned. In the following, we call the group-by query that computes the sorted list of scores for an object space a "**probe**" query, which can be implemented as follows. Given an object space $\{D'_1, D'_2, \ldots\}$, the probe query is implemented as:

`select` $M(Measure)$
`from` fact table
`group by` $D'_1, D'_2, \ldots$
`order by` $M(Measure)$ `desc`.

Since this query may involve a full scan of the whole data set in the worst case, the probe query is very costly. Therefore, the overall query execution cost is dominated by the probe queries. More formally, supposing that the probe query incurs a scan of the data with the cost $Cost_{TS}$, the overall query processing cost can be expressed as $O(L \times Cost_{TS})$ since there are totally L probes. This is prohibitive for large data sets because (1) L grows exponentially with respect to d; and (2) $Cost_{TS}$ is expensive, and it is difficult to share the aggregation across different object spaces due to our assumption that M can be an arbitrary aggregate function.

5.3.2 Full Materialization

Alternatively, one can simply fully materialize all sorted lists. Then the online query execution can be done in two steps. First, we compute $C(p)$ by selecting and aggregating all base tuples matching p. The score of each descendent object $q \in C(p)$ can be derived using those base tuples. In the second step, for each descendent object $q \in C(p)$ we look up the materialization to obtain $Rank(q)$. Finally, the top-k objects with the smallest $Rank$ can be outputted.

For the online cost, the first step depends on the selectivity of p, which is usually low and can be further sped up through auxiliary index structures. The second step incurs trivial cost. Unfortunately, the storage cost of such a full materialization strategy is infeasible for large data sets due to the "curse of dimensionality" [35]. Also, there could be more than one measure dimension, so the total storage space would be proportional to the number of measure dimensions multiplying the total number of objects in the data set.

5.3.3 Horizontal Strategy

Given a storage budget, how can we seek a middle ground between query execution efficiency and the storage space? In response to this query, we propose a simple *horizontal score index* strategy. Intuitively, we can store some highly ranked scores, which can avoid some computation during the online phase.

Definition 18 (Horizontal Score Index) *Given a storage budget ratio α ($0 \leq \alpha \leq 1$), the* horizontal score index *maintains for each object space S_l its top-α scores, i.e., it maintains the list $\{m_1^l, m_2^l, \ldots, m_{\lfloor |S_l| \times \alpha \rfloor}^l\}$ for $1 \leq l \leq L$.*

This strategy precomputes the top-α portion of all sorted lists. Correspondingly, the query execution consists of the following steps. First, same as for full materialization, all descendent objects and their scores are computed. Second, all the scores are checked against the index. If k or more scores are *seen* (*i.e.*, appears) in the index, the exact top-k objects can be returned. Otherwise, because less than k scores are seen, we would not be able to derive the exact top-k from the index. Consequently, we must probe all object spaces with at least one unseen score. Thus, the query execution cost of the horizontal strategy is either very small (*i.e.*, no probe query) or very

high (*i.e.*, all object spaces are probed).

Setting the only parameter α to 0 or 1 would make this approach degenerate to no or full materialization, respectively. When k is often small and the top-k objects tend to be highly ranked, this ratio may be set to a small value (*e.g.*, 0.1%) to get a good balance.

It is worth mentioning that a quantile-like structure, as discussed in the previous chapters, would not work well for the kOSearch problem. The reason is the following. Since the query object may have *multiple* descendent objects in each object space, a probe query needs to be executed for an object space unless *all* descendent objects in that space are pruned. Using a quantile-like structure, one may be able to bound the rank for *some* descendent objects in an object space, but in many cases it is unlikely that *all* objects can be pruned. Thus, a promotion cube like structure will not perform well since it could often degenerate to the no materialization strategy. Moreover, since an exact query execution algorithm requires us to output the exact rank for each top-k object, a quantile-like structure may not be helpful since it cannot help derive the exact ranks.

5.3.4 Vertical Strategy

The horizontal score index has unpredictable performance because it may often degenerate to the no materialization strategy. To address the problem, we propose a more robust strategy called the vertical score index, defined as follows.

Definition 19 (Vertical Score Index) *Given a storage budget ratio α ($0 \leq \alpha \leq 1$), the* vertical score index *maintains the sorted list of scores for the L' ($0 \leq L' \leq L$) smallest object spaces such that their total size does not exceed $\alpha \times \sum_l |S_l|$.*

Here we call an object space S_l small if it contains few objects (*i.e.*, $|S_l|$ is small). For query execution, precisely those non-materialized object spaces will incur probe queries. The advantage of this strategy over the horizontal strategy is that we can guarantee no probe query is executed for the materialized spaces. Moreover, it can be proven that this strategy is optimal in terms of robustness in the sense that, given a storage budget α, any other exact strategy would require no less than $L - L'$ probes in the worst case.

Algorithm 5.1: Exact query execution

/* **pruning** */
1: *QueryTupleSet* $\leftarrow$ retrieve all base tuples of the fact table that matches the input query object p;
2: Recursively aggregate *QueryTupleSet* in order to compute $C(p) \leftarrow \{c_1, c_2, \ldots, c_n\}$ and their scores $\{M(c_1), M(c_2), \ldots, M(c_n)\}$;
3: **for** $i \leftarrow 1$ **to** n **do**
4: Check the score $M(c_i)$ against the materialized sorted list, if any, of object space $S(c_i)$;
5: **if** $M(c_i)$ is seen
6: $HRank(c_i) \leftarrow LRank(c_i) \leftarrow c_i$'s exact rank;
7: **else**
8: Set $HRank(c_i)$ and $LRank(c_i)$;
9: **end**
10: $\gamma \leftarrow$ the k-th highest $LRank(c_i)$;
11: $C' \leftarrow \{c_i | HRank(c_i) \leq \gamma \wedge HRank(c_i) \neq LRank(c_i)\}$;
/* **probing** */
12: **foreach** $S_l \in \mathbb{S}$ that contains ≥ 1 object in C' **do**
13: Probe S_l;
14: Get the ranks for objects in C' contained in S_l;
15: **end**
16: Return $C^k(p)$, the top-k objects with exact ranks;

Table 5.2: The general exact algorithm.

5.3.5 General Query Execution Algorithm

The pseudo code for the general exact query execution algorithm is displayed in Table 5.2. It is divided into two phases, a pruning phase (Lines 1–11) and a probing phase (Lines 12–16). In the former phase, the query object's descendent objects are aggregated from *QueryTupleSet*, the set of raw tuples matching the query. Given the n descendent objects and their aggregate scores, we check against the materialized index, if any, to derive a highest possible rank (*HRank*) and lowest possible rank (*LRank*) for each object. If for a descendent object c_i ($1 \leq i \leq n$), its score $M(c_i)$ is seen in the sorted list, we can obtain its exact rank (Lines 5–6). Otherwise, its lowest possible rank, $LRank(c_i)$, will be set to 1.0 (Line 8), whereas $HRank(c_i)$ will depend on the indexing strategy. Specifically, for the horizontal strategy, we have $HRank(c_i) \leftarrow \frac{\lfloor |S(c_i)| \times \alpha \rfloor + 1}{S(c_i)}$, and for others $HRank(c_i) \leftarrow \frac{1}{|S(c_i)|}$ (Line 8). Then, a rank threshold γ is computed as the k-th highest *LRank*, which means that any top-k object should be ranked no lower than γ (Line 10). The candidate object set, C', is computed as the "promising" objects whose exact rank is unknown (Line 11).

In the probing phase, all descendent objects which can potentially be in the top-k results are probed for their exact ranks (Lines 12–15). Note that any object space containing at least one object from the candidate object set C' needs to be probed (Line 12). The overall cost of this algorithm is dominated by the probe query (Line 13). Finally, the top-k descendent objects along with their exact ranks can be outputted (Line 16).

5.4 A Probabilistic Approximate Framework

The horizontal and vertical score indices can facilitate exact query processing. However, they may not yield a satisfactory tradeoff for very large data sets, especially in the presence of high dimensionality. This motivates us to develop a *probabilistic score index* (psIndex) framework for approximate query answering. Our goal is to (1) substantially reduce the index size, and (2) significantly improve the query performance.

We observe that the performance can be largely improved when the exactness constraint of the top-k answers is relaxed with very little or even no sacrifice in the result quality, which is often acceptable for explorative applications. Now, each kOSearch query is parameterized by a user-

Strategy	Storage	Probe
No materialization	0	L
Full materialization	prohibitive	0
Horizontal	α	sometimes efficient
Vertical	α	$L - L'$
Probabilistic	$\ll \alpha$	efficient

Table 5.3: A roadmap of different strategies studied.

specified probabilistic confidence threshold θ ($0 \leq \theta \leq 1$) to control the balance between the query response time and the precision of top-k answers. Particularly, when $\theta = 1.0$, it would guarantee the top-k objects outputted to be completely accurate. When, on the other hand, $\theta = 0.0$, the pruning power can be maximized and no probing would be needed at all such that the query execution becomes extremely efficient. In practice, a reasonably large threshold may yield a good tradeoff between efficiency and accuracy.

This probabilistic framework consists of two orthogonal components. The first component contains *piecewise regression functions* to model the sorted lists of scores at the offline stage (Sections 5.4.1). The second component answers top-k queries at the online phase by leveraging the regression models and their residual error distributions for conducting *probabilistic pruning* (Sections 5.4.2 and 5.4.3). We defer detailed discussions of these components to Sections 5.4.4 through 5.4.7.

Table 5.3 outlines a summarization of all the five methods studied in this chapter. We can see that the no materialization and full materialization strategies incur the maximum query execution cost (characterized by the number of probe queries) and storage space, respectively. The horizontal and vertical strategies can improve the query performance given some storage budget. The pslndex strategy studied in this section provides the best tradeoff between storage space and efficiency as its storage requirement could be much smaller than the exact strategies, while achieving significant better online performance.

5.4.1 Probabilistic Score Index Structure

In this subsection, we develop the probabilistic score index (pslndex) structure. Unlike the exact score materialization, pslndex approximates the score distributions using piecewise regression models.

Definition 20 (Probabilistic Score Index) *Suppose a storage budget α ($0 \leq \alpha \leq 1$) is given. For each object space S_l ($1 \leq l \leq L$), we materialize a piecewise regression model (PRM) for its sorted list of scores consisting of*

- $(g+1)$ knots $\{m^l_{r_0}, m^l_{r_1}, \ldots, m^l_{r_g}\}$;
- g regression functions $\{f^l_0, f^l_1, \ldots, f^l_{g-1}\}$, *where*

$$f^l_i : \{r_i, r_i+1, \ldots, r_{i+1}\} \rightarrow \mathbb{R};$$

- g *residual* error *parameters* $\{\sigma^l_1, \sigma^l_2, \ldots, \sigma^l_g\}$.

Note that g is a non-negative integer derived from α.

We elaborate on this definition in the following and defer further details to Section 5.4.4. For an object space S_l, we select scores from $(g+1)$ positions $\{r_0, r_1, \ldots, r_g\}$ as the "knots". Here these positions are assumed to be evenly spaced, although their selection can be further optimized to maximize the efficiency (see Section 5.4.4). Then, for each of the g segments of scores separated by the knots, we learn a linear function

$$f^l_i(r) = \hat{\beta}^l_i r + \hat{\xi}^l_i$$

using the linear least squares method:

$$\begin{aligned} \hat{\beta}^l_i &= \frac{(r_{i+1} - r_i + 1)\sum_{r=r_i}^{r_{i+1}} r m_r - (\sum_r m_r)(\sum_r r)}{(r_{i+1} - r_i + 1)\sum_r r_i^2 - (\sum_r r)^2}, \\ \hat{\xi}^l_i &= \frac{\sum_r m_r - \hat{\beta}_i \sum_r r}{r_{i+1} - r_i + 1}. \end{aligned}$$

While there exists a broad spectrum of regression models (*e.g.*, spline functions) that can be applied in our problem context [52], we adopt the linear least squares approach for its conciseness of representation and low computational complexity. Given the linear functions learned, we will be able to estimate the rank for any given score and further compute the residual errors to quantify the quality of the rank estimation.

Rank estimation: Given any object space S_l and any object q in S_l with score $M(q)$. We can compute q's estimated rank as follows. Suppose $M(q)$ lies in the range $[m_{r_i}, m_{r_{i+1}}]$ for the smallest possible i ($\in [0, g-1]$), we will have the estimated rank of q:

$$\hat{Rank}(q) = \begin{cases} \dfrac{M(q) - \hat{\xi}_i}{\hat{\beta}_i |S_l|} & \text{if } r_i \leq \frac{M(q)-\hat{b}_i}{\hat{\beta}_i} \leq r_{i+1}; \\ \dfrac{r_i}{|S_l|} & \text{if } \frac{M(q)-\hat{b}_i}{\hat{\beta}_i} < r_i; \\ \dfrac{r_{i+1}}{|S_l|} & \text{if } \frac{M(q)-\hat{b}_i}{\hat{\beta}_i} > r_{i+1}. \end{cases}$$

Residual error: Clearly, it would be desirable for the regression models to have as smaller a difference between $\hat{Rank}(q)$ and $Rank(q)$ as possible. An ideal scenario would be that the estimation is completely accurate, so the online performance would be as good as the full materialization approach. In practice, since the scores are monotonically decreasing, a piecewise regression model often yields good approximation given a reasonable g value. In the unlikely event that the scores have many outliers, one may resort to the least absolute deviations or quadratic functions [52, 70] to learn the models. We can safely assume that the resulting error follows the Gaussian distribution for large data sets [22]. More specifically, for a segment $[r_i, r_{i+1}]$ in object space S_l, we assume that the rank estimation error follows the Gaussian distribution $\mathcal{N}(0, (\sigma_i^l)^2)$, where σ_i^l is the standard deviation of the residual error distribution. The mean of the residual error distribution is assumed to be 0, since this can be easily achieved by perturbing the estimated parameter $\hat{\xi}_i^l$. Now, denote by $q_{r_i}, q_{r_i+1}, \ldots, q_{r_{i+1}}$ the objects in segment $[r_i, r_{i+1}]$, and using the maximum likelihood estimation we can derive the standard deviation $\sigma_i^l = \sqrt{\frac{1}{r_{i+1}-r_i+1} \sum_r (r - \hat{Rank}(q_r))^2}$. This way, the residual error distribution parameters for all segments can be computed.

Computational complexity: Let us discuss the overall computational complexity of constructing the pslndex given the L sorted lists. We can see that learning the linear functions, which is linear with respect to the number of scores, takes $O(\sum_l |S_l|)$ time. The maximum likelihood estimation of the error distribution parameters also takes $O(\sum_l |S_l|)$ time. Thus the overall time cost of building the probabilistic score index is *linear* with respect to the total size of the sorted lists. For space complexity, we can see that the whole process uses only $O(1)$ intermediate space.

5.4.2 Query Execution with Quality Guarantee

Given psIndex, any kOSEARCH query can be parameterized by the following: (i) p, the query object, (ii) a non-negative integer k, and (iii) θ, a confidence threshold within the range $[0, 1]$, which we will discuss shortly. The goal of the query execution algorithm is to avoid computing as many sorted score lists as possible using the given psIndex. It works in three steps. First, we generate a set of *boundary* objects that are the candidate top-k results. That is, for each object in the boundary set, its membership in the top-k results is unclear; for any object that is not in the boundary set, we are certain that either it is a top-k object or it is not. Second, we conduct probabilistic pruning to further shrink the size of the boundary object set by exploiting the following observation: when a boundary objects p_1 has a much higher *estimated* rank than another boundary object p_2 does, it is very unlikely that p_1 is *actually* ranked lower than p_2, and vice versa. Therefore, any boundary objects which are either highly or low ranked in a probabilistic sense can be pruned from the boundary object set. Third, we probe the true ranks for all the remaining boundary objects, and then return the top-k answers.

Step 1 (Boundary object generation): The goal of this step is to compute a small set of boundary objects whose top-k membership is unclear. To begin with, we compute the query object p's descendent objects, $C(p) = \{c_1, c_2, \ldots, c_{|C(p)|}\}$ and their scores by accessing p's tuples. For each of p's descendent objects $c \in C(p)$, we can derive c's best possible rank $\mathit{HRank}(c)$ and worst possible rank $\mathit{LRank}(c)$ by comparing c's score $M(c)$ with the knots in the probabilistic index. After computing all objects, let LRank_{thres} be the k-th best $\mathit{LRank}(\cdot)$. Then, all descendent objects $c \in C(p)$ satisfying $\mathit{HRank}(c) > \mathit{LRank}_{thres}$ (*i.e.*, unpromising) will be pruned because they can never be in the top-k answers. Conversely, let LRank_{thres} be the k-th best $\mathit{HRank}(\cdot)$. All descendent objects $c \in C(p)$ satisfying $\mathit{LRank}(c) < \mathit{HRank}_{thres}$ must be in the top-k answers (and we call them promising objects). Suppose that there are k_1 such objects and we have $k_1 < k$. Excluding the promising and unpromising objects, the remaining objects are called *boundary* objects whose top-k membership is unclear. We denoted the set of boundary objects by $B(p) = \{b_1, b_2, \ldots, b_n\}$ ($n = |B(p)|$, $B(p) \subseteq C(p)$), where n is the number of boundary objects. Letting $k' = k - k_1$, our goal becomes to compute the top-k' boundary objects because there are already k_1 objects which must be in the top-k results.

Step 2 (Probabilistic pruning): In this step, we conduct probabilistic pruning to further prune objects from the boundary object set which are very likely or unlikely to be in the top-k results. Since our goal here is to select the top-k' objects from the n boundary objects, the idea is that for each boundary object b_j ($1 \leq j \leq n$), we will compute the probability that it is in the top-k' of the boundary object set $B(p)$:

$$\Pr\{b_j \text{ is in the top-}k'\}, \tag{5.1}$$

which we denote by $\Pr(j)$ for simplicity. For any boundary object b_j, $\Pr(j) = 1.0$ means that b_j is certainly in the top-k', whereas $\Pr(j) = 0.0$ means the contrary. We will defer the discussion of how to compute $\Pr(j)$ to Section 5.4.3. Now, given the confidence threshold θ, we can partition $B(p)$ into three disjoint subsets based on the probabilities:

- $B^+ = \{b_j \in B(p) | \Pr(j) \geq \theta\}$: this subset consists of all boundary objects that are top-k' with confidence at least θ;
- $B^- = \{b_j \in B(p) \setminus B^+ | \Pr(j) \leq 1 - \theta\}$: this subset consists of boundary objects that are not top-k' with confidence at least θ; note that this subset is disjoint with B^+ even when the confidence threshold is low (*e.g.*, < 0.5); and
- $B^c = B \setminus (B^+ \cup B^-)$: all the remaining objects.

Intuitively B^+ is the set of boundary objects which are in the top-k' with high probability, whereas B^- contains objects which are not in the top-k' with high probability. Because θ encodes the confidence level required by the user, we can *append* the objects in B^+ into the final list of top-k objects and prune those in B^-.

Step 3 (Probing): After the above two steps, we have generated k_1 objects that must be in the top-k results and $|B^+|$ (*i.e.*, the number of objects in B^+) objects that are in the top-k with a confidence level passing the user-specified threshold θ. To produce the final top-k answers, two cases need to be considered. First, when $k_1 + |B^+| < k$, we probe all the object spaces that contain at least one object in B^c and then compute the top-$(k - k_1 - |B^+|)$ objects in B^c. We then *append* these objects to the final top-k list, which is subsequently returned to the user, and the

query execution is done. Second, when $k_1 + |B^+| \geq k$, we can simply append the $k - k'$ objects in B^+ with the highest probabilities to the final top-k list. Ties are broken arbitrarily. In this case, no object space needs to be probed. This is reasonable because this case often happens when the confidence threshold is low.

Analysis: Given a PSIndex with reasonable size, we often have $|B^c| \ll |B(p)| \ll |C(p)|$. Because only object spaces that contains at least one boundary object in $|B^c|$ will be probed, the number of probe queries needed will be much less than the total number of object spaces, especially in the presence of a large number of object spaces. Since the total query execution cost is dominated by the probing step, the overall probabilistic query execution cost will be dramatically smaller and more robust than the cost of any previous exact algorithms. Meanwhile, the confidence parameter is able to offer a controllable balance between accuracy of the top-k results and query execution efficiency.

5.4.3 A Random Sampling Method for Probability Computation

Let us turn to the problem of computing $\Pr(j)$ for the boundary objects in Equation (5.1). For any $b_j \in B(p)$ $(1 \leq j \leq n)$, its true rank $Rank(b_j)$ can be considered as an addition of a Gaussian noise to its estimated rank $\hat{Rank}(b_j)$. The estimated rank $\hat{Rank}(b_j)$ can be derived using the linear regression function corresponding to the segment object b_j belongs to. The Gaussian noise, on the other hand, encodes the random noise introduced by the residual error of the regression function. Mathematically, we have

$$\begin{aligned} Rank(b_j) &\sim \hat{Rank}(b_j) + \mathcal{N}(0, (\sigma_i^l)^2) \\ &\sim \mathcal{N}(\hat{Rank}(b_j), (\sigma_i^l)^2), \end{aligned}$$

where σ_i^l is the materialized residual error parameter (*i.e.*, the standard deviation of the Gaussian distribution) for b_j's segment. Thus, for a particular boundary object $b_j \in B(p)$, $Rank(b_j)$ can be considered as a random variable following the Gaussian distribution, and the n boundary objects's rank, $Rank(b_1), Rank(b_2), \ldots, Rank(b_n)$, follow n independent, continuous distributions.

The meaning of $\Pr(j)$ for any j can be interpreted using the possible world semantics: given the

n randomly distributed ranks, there are exactly $n!$ possible worlds, $\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_{n!},$, each encodes a unique ordering of $b_1, b_2, \ldots, b_n$ (ordered by $Rank(\cdot)$). For instance, given $n = 3$, there are $n! = 6$ possible rankings of the 3 objects. Based on such a possible world interpretation, we can compute $\Pr(j)$ for each $1 \leq j \leq n$ using the following equation:

$$\Pr(j) = \sum_{\substack{\mathcal{W}_i : 1 \leq i \leq n! \\ b_j \text{ is in the top-}k' \text{ in } \mathcal{W}_i}} \Pr(\mathcal{W}_i), \tag{5.2}$$

where $\Pr(\mathcal{W}_i)$ $(1 \leq i \leq n!)$ denotes the probability of the i-th possible world, or in other words, the probability that the true rank ordering matches $\mathcal{W}_i$. Because these possible worlds are independent, the probability that b_j belongs to the top-k' is equal to the sum of the probabilities of all possible worlds where b_j is ranked within top-k'. Unfortunately, it would be simply impossible to derive $\Pr(j)$ directly using Equation (5.2) because (i) we need to enumerate all possible worlds and the number of possible world can be extremely large when n is large; and (ii) computing $\Pr(\mathcal{W}_i)$ for any $\mathcal{W}_i$ involves n-dimensional integral of continuous distributions, which is not possible.

To solve this problem, we use a random sampling method as follows. We independently randomly generate an *n-dimensional vector* following the distribution of $(Rank(b_1)Rank(b_2)\cdots Rank(b_n))$ for N times. Each of these N vectors contain n values, which are independently drawn from the distribution $Rank(b_1), Rank(b_2), \ldots, Rank(b_n)$, respectively. When generating these N vectors, we keep track of the number of occurrences of each object appearing in the top-k' and compute:

$$\hat{\Pr}(j) = \frac{\text{the number of vectors where } b_j \text{ is in top-}k'}{\text{total number of vectors sampled: } N}. \tag{5.3}$$

For example, if $n = 3$, $k' = 2$, and a sample vector generated is [2.5 1.0 3.3], we will increment the counter for b_2, which is ranked to top-1 in this vector. This way, the frequency ratios $\hat{\Pr}(j)$ will serve as an estimation of the true probability values $\Pr(j)$ for $1 \leq j \leq n$. In comparison to a brute-force method that directly computes the probabilities, this random sampling method has several advantages: (i) it is computationally feasible, (ii) it guarantees that $\hat{\Pr}(j)$ is very close to its true value when the number of samples N is sufficiently large, and (iii) this guarantee of quality is independent of the vector dimensionality n, which is desirable because the number of boundary

objects is usually large. We formally state the quality guarantee of this method in the following claim.

Claim 3 *The sampling error* $\Pr\{|\hat{\Pr}(j) - \Pr(j)| \geq \epsilon\}$ *decays linearly with* $\frac{1}{\sqrt{N}}$*, where* ϵ *is any small constant and* N *is the total number of vectors sampled.*

Proof: This can be proved using the Central Limit Theorem [67]. We defer the details of the proof to Section 5.4.6. ∎

Computational complexity: To compute $\hat{\Pr}(j)$ for all $1 \leq j \leq n$ using random sampling, we initially maintain a hash table keyed on j to count in how many vectors b_j appears in the top-k'. We assume that the cost of generating a one-dimensional Gaussian sample value is constant. Every time we generate an n-dimensional sample vector, we use a quicksort-like algorithm (*i.e.*, to select the top-k' from n elements) to derive the top-k' boundary objects in $O(n + k' \log k')$ time and then increment the counter for them. Therefore, the total computation process for obtaining all these estimated probabilities (and thus the probabilistic pruning step) takes $O(nN + k'N \log k')$ time. The space complexity is $O(n)$ because only a hash table and a sample vector need to maintained. In some cases, we can quickly remove some objects which are obviously likely or unlikely to be in the top-k results. Section 5.4.7 further presents an upper bound-based pruning strategy to speed up the probabilistic pruning.

Note that the cost of the random sampling method and the probabilistic pruning step can be ignored when comparing to the probing cost. In our empirical evaluation, we found that this process is very fast with very large values of N and n.

5.4.4 The Offline Algorithm

In this subsection, we present the pseudo code for the psIndex offline algorithm and clarify several issues.

psIndex construction: Table 5.4 shows the pseudo code for the offline algorithm. The outer loop iterates over all object spaces and the inner loop goes through each segment of the piecewise regression function. Each inner loop (Lines 5–10) runs in linear time, $O(|S_l|)$, so the overall algorithm is efficient.

Scalability of index construction: By sequentially aggregating the sorted lists, the algorithm is scalable to large data by accessing them in a streaming fashion.

Selection of knots: Note that the positions of the knots, $r_0, r_1, \ldots, r_g$, are taken as input parameters. In this algorithm, we select a set of uniformly spaced knots. In principle, however, other non-uniform selections are possible to optimize the efficiency. First, one may select the knots to minimize the global error of the piecewise regression model. Second, a heuristic approach is to reduce the local regression error by selecting denser knots for certain segments. Such selection can be learned through user queries, where certain segments tend to contain boundary objects more frequently than others. However, these questions are beyond the scope of this chapter and are left for future study.

Parameter g: In our linear model, each object space stores $4g + 1$ values. Thus, given α, we set g to the largest integer s.t. $(4g + 1)L + g + 1 \leq \alpha \times \sum_l |S_l|$.

5.4.5 The Online Algorithm

The online algorithm in Table 5.5 is organized by the three steps: boundary object generation (Lines 1–8), probabilistic pruning (Lines 9–16), and probing (Lines 17–19). Note that at Line 15, we require that $|B^+| \leq k' = k - |C(p)|$ (otherwise there would be more than k answers) and $|B^c| \leq |B(p)| - k'$ (otherwise there will fewer than k answers). When B^+ is too large (*i.e.*, $|B^+| > k'$), we can keep only the objects with the top-k' largest probabilities in B^+. Similarly, we can reduce the size of B^- when it contains too many objects.

Confidence threshold θ: When a user sets $\theta = 1.0$, she desires the top-k to be precise. In such cases, we replace the probabilistic pruning phase (Lines 9–16) with the following: compute for each boundary object how many other boundary objects are ranked absolutely higher or lower (note that the residual error of the regression of any segment is finite). Then, promising and unpromising boundary objects (*i.e.*, B^+ and B^-) are identified in a deterministic fashion instead of relying on the sampling method.

Computational complexity: We break down the cost by step:

- The boundary object generation step can be efficiently processed in $O(|C(p)| \log |C(p)|)$ time and uses $O(|C(p)|)$ space;

Algorithm 5.2: Probabilistic index construction

```
1:  foreach S_l ∈ 𝕊 do
2:      Probe S_l to obtain the sorted list of scores
3:      {m^l_1, m^l_2, ..., m^l_|S_l|};
4:      Store (g + 1) knots {m^l_{r_0}, m^l_{r_1}, ..., m^l_{r_g}};
5:      for i ← 0 to g − 1 do
6:          Compute linear least squares regression for
            segment {m^l_{r_i}, m^l_{r_i+1}, ..., m^l_{r_i+1}};
7:          Store β^l_i and ξ^l_i;
8:          Compute the variance of the regression;
9:          Store (σ^l_i)^2;
10:     end
11: end
```

Table 5.4: The offline algorithm for psIndex.

- The cost of the probabilistic pruning step has been partially discussed in Section 5.4.3, which takes $O(nN + k'N \log k') \leq O(nN + kN \log k)$ time and $O(n)$ space;
- The last probing step takes $O(|B^c| \times Cost_{TS})$ time, where $Cost_{TS}$ denotes the table scan cost, and $O(1)$ space. This is the efficiency bottleneck.

Overall, the algorithm has time complexity $O(|B^c| \times Cost_{TS})$ and space complexity $O(|C(p)|)$. This complexity result justifies the key thrust of the probabilistic strategy: by pruning many unlikely boundary objects, $|B^c|$, the crucial factor in query cost, can be minimized.

5.4.6 Proof of the Sampling Quality

Now we formally prove Claim 3. Suppose N sample n-dimensional vectors, $V_1, V_2, \ldots, V_N$ are drawn from some independent Gaussian distributions. Let a random vector $\bar{v}$ represent the joint distribution (the domain of $\bar{v}$ is the n-dimensional space $\mathbb{R}^n$). Let $\mathcal{I}_j$ $(1 \leq j \leq n)$ be an identity function such that $\mathcal{I}_j(\bar{v}) = 1$ if b_j is in top-k' according to $\bar{v}$; or 0 otherwise. Thus, $\Pr(j) = E[\mathcal{I}_j(\bar{v})]$. Also, $\hat{\Pr}(j) = \sum_{i=1}^{N} \mathcal{I}_j(V_i)/N$ by Equation (5.3).

By Central Limit Theorem [67], when N is sufficiently large, we have $\overline{V}_j = \sum_{i=1}^{N} \mathcal{I}_j(V_i)/N$ follows Gaussian distribution $\mathcal{N}(E[\mathcal{I}_j(\bar{v})], \sigma^2_{\mathcal{I}_j(\bar{v})}/N)$, where $\sigma^2_{\mathcal{I}_j(\bar{v})}$ is the variance of $\mathcal{I}_j(\bar{v})$. Therefore, when we quadratically increase N, the standard deviation $\sqrt{\sigma^2_{\mathcal{I}_j(\bar{v})}/N}$ will be linearly decreasing; consequently, fixing any small ϵ (*e.g.*, 0.001), the (error) probability that $\hat{\Pr}(j)$ deviates more than

ϵ from the mean, $\Pr(j)$, will be linearly decreasing. Clearly, this relationship is independent of n, the dimensionality of the vectors.

5.4.7 An Upper Bound-based Pruning Strategy

Intuitively, a boundary object is likely to be in the top-k' when (i) the boundary object b_j is ranked higher than other boundary objects and (ii) the "estimated rank gap" between them is large. The reverse also holds for an unlikely top-k' object. In this subsection, we exploit this intuition and develop an upper bound-based strategy for probabilistic pruning. It can complement the sampling method (Section 5.4.3) in the presence of a low confidence threshold θ since such upper bound computation is extremely efficient. The method is based on the following claim:

Claim 4 *Given any two boundary objects b_j and b_l so that $\hat{Rank}(b_j) < \hat{Rank}(b_l)$, $Rank(b_j) \sim \mathcal{N}(\hat{Rank}(b_j), \sigma_j^2)$, and $Rank(b_l) \sim \mathcal{N}(\hat{Rank}(b_l), \sigma_l^2)$, we have*

$$\Pr\{Rank(b_j) > Rank(b_l)\} \leq \frac{Var[Y]}{(\hat{Rank}(b_l) - \hat{Rank}(b_j))^2}, \tag{5.4}$$

where $Var[Y]$ is the variance of the auto-correlation [67] of two Gaussian distributions $\mathcal{N}(0, \sigma_j^2)$ and $\mathcal{N}(0, \sigma_l^2)$.

Proof: This inequality can be derived using Chebyshev's inequality [67] and the fact $E[Y] = 0$. ∎

Supposing we have materialized $Var[Y]$ for all pairs of residual error distributions, then Equation (5.4) can be computed in $O(1)$ time for any pair of boundary objects b_j and b_l. Now, the probabilistic pruning of boundary objects works as follows (*i.e.*, replace Table 5.5 Lines 9–15 with the following): For each object having the top-k' estimated rank, we compute its probability of "being swapped out of the top-k'" using Equation (5.4). If this probability is lower than $1 - \theta$, we will add it into B^+. Similarly, for the remaining object, we compute its probability of "being swapped into the top-k'". If this probability is lower than $1 - \theta$, we will add it into B^-. Then, let $B^c \leftarrow B \setminus (B^+ \cup B^-)$.

This process has time complexity $O(n^2)$, which is more efficient than the sampling method. Due to the limited space, we do not report its performance result in the experiments.

Algorithm 5.3: Probabilistic query processing

/* **boundary object generation** */
Retrieve *QueryTupleSet*;
Aggregate *QueryTupleSet* to compute $C(p)$;
foreach $c \in C(p)$ **do**
    Compute $HRank(c)$ $LRank(c)$, and $\hat{Rank}(c)$;
end
Compute $LRank_{thres}$ and $HRank_{thres}$;
$C^k(p) \leftarrow \{c | LRank(c) < HRank_{thres}\}$;
$B(p) \leftarrow \{c | HRank(c) < LRank_{thres}\} \setminus C^k(p)$;
/* **probabilistic pruning** */
Initialize a counter hash table for each $b_j \in B(p)$;
repeat N times
    Randomly generate an n-dimensional vector;
    Increment the counter of the top-k' objects according to the vector;
end
$\hat{\Pr}(j) \leftarrow \frac{b_j\text{'s counter}}{N}$ for each $1 \leq j \leq n$;
Partition $B(p)$ to B^+, B^-, and B^c using θ;
Append B^+ to $C^k(p)$;
/* **probing** */
Probe all object spaces containing ≥ 1 object in B^c;
Append the top-$(k - |C^k(p)|)$ objects in B^c to $C^k(p)$;
Return $C^k(p)$ with the objects' estimated ranks;

Table 5.5: The online algorithm for psIndex.

Jennifer Widom		*Christos Faloutsos*	
Query	Rank	Query	Rank
(*SIGMOD*)	0.08%	(*self*)	0.01%
(*VLDB*)	0.10%	(*VLDB*)	0.05%
(*self*)	0.11%	(*KDD*)	0.05%
(*ICDE*)	0.42%	(*2004*)	0.06%
(*2000*)	0.42%	(*KDD, 2008*)	0.06%

Table 5.6: Top-5 objects of two example queries.

5.5 Experiments

In this section we evaluate the effectiveness of the kOSearch query and the efficiency of our proposed algorithms. We first conduct a DBLP [1] case study (Section 5.5.1), followed by a comprehensive performance study on the TPC-H decision support benchmark [3] (Section 5.5.2). Our findings can be summarized as follows:

- kOSearch queries generate meaningful results in our case study;
- The probabilistic approach consistently outperforms the horizontal and vertical approaches by 1 to 2 orders of magnitude in query processing efficiency, while using only 1% of their storage overhead;
- The precision of the probabilistic approach is 100% in most cases, and is in general above 99%.

All our experiments were done on a computer with 2.5GHz dual-core CPU, 4GB of memory, and 250GB of hard disk. The source code was compiled in Microsoft Visual C# and the data was stored in Microsoft SQL Server 2008 running in Windows XP.

5.5.1 A DBLP Case Study

The DBLP data set contains about $1.7M$ base tuples, each corresponding to a paper with the three dimensions: *Author, Venue*, and *Year*. Four object spaces are considered: {*Author*}, {*Author, Year*}, {*Author, Venue*}, and {*Author, Venue, Year*}. *COUNT* is used to rank objects. Table 5.6 displays the top-5 results for two query authors (the *Author* dimension is omitted). As expected, these results match our intuition well.

5.5.2 Performance

TPCH data: For performance study, we generate a TPCH [3] fact table with $6M$ tuples and $d = 10$ dimensions. We choose $L = 120$ object spaces, each having no more than $5M$ objects and we use SUM as the default aggregate function. The standard TPCH benchmark generator [3] is used to generate a *lineitem* table containing $6M$ base tuples. The following 10 dimensions are used: *suppkey* (cardinality=10000), *linenumber* (7), *quantity* (50), *discount* (11), *tax* (9), *returnflag* (3), *linestatus* (2), *shipdate* (2526), *shipinstruct* (4), and *shipmode* (7). All object spaces with more than $5M$ objects are ignored.

Algorithms for comparison: Five methods are compared: Full (full materialization), Empty (no materialization), Horizon (horizontal score index), Vertical (vertical score index), and Prob (probabilistic score index). For Prob, we set the default number of samples $N = 10^5$ and set the confidence threshold $\theta = 0.8$.

Query execution time and *storage overhead* are the performance metrics. For Prob, its precision will be analyzed as well: given a top-k query, the *precision* is defined as the number of true top-k objects returned divided by k.

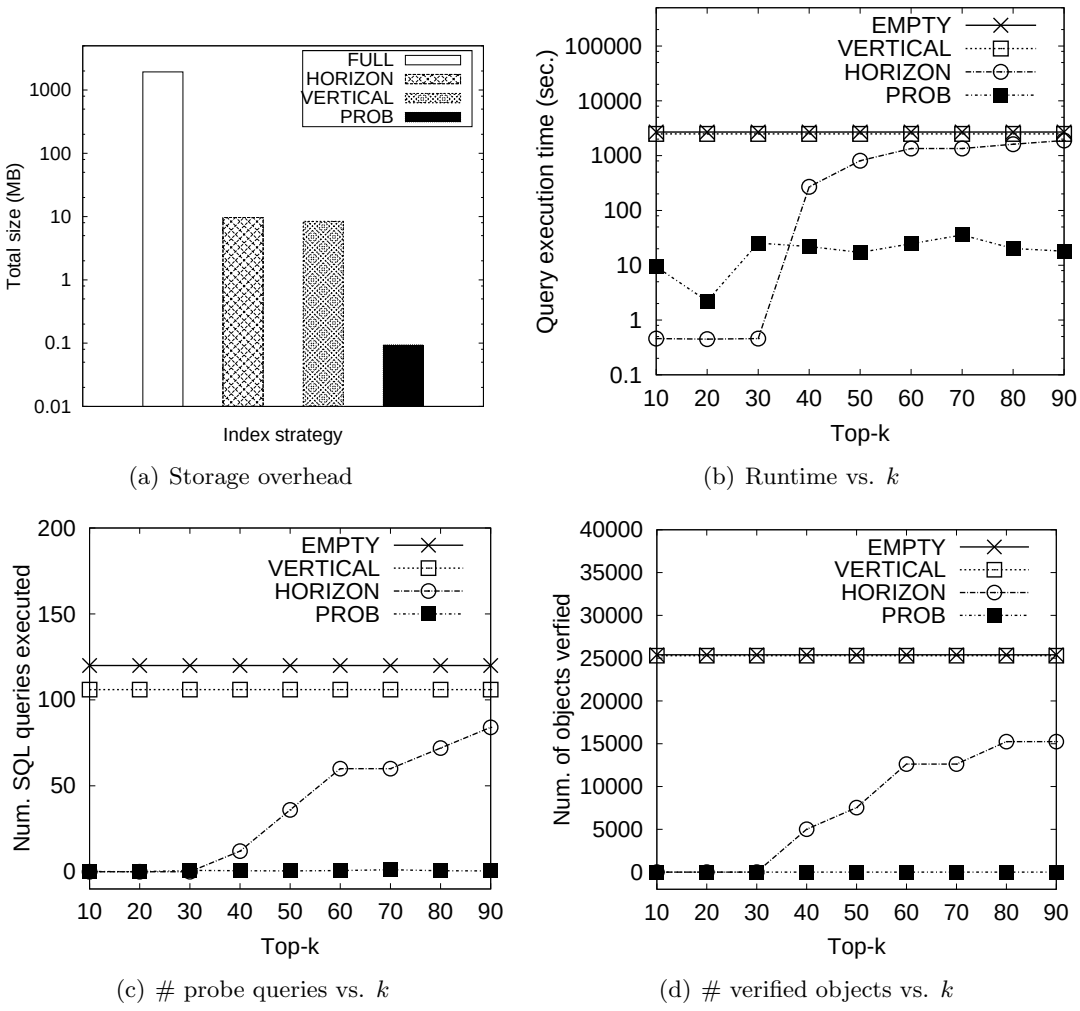
Query workload: 10 distinct objects are uniformly randomly generated from the object space $\{suppkey\}$. All results are averaged over these objects.

5.5.3 Storage

Figure 5.3(a) displays a comparison of the storage overhead of the strategies. Full consumes nearly 2GB of disk space, whereas for Vertical and Horizon we set the storage budget ratio α to 0.5% and they use about 9MB of space. For Prob, we set α to 0.005% and its size is only 95KB, about 1% of Horizon and Vertical. Thus, the probabilistic score index is orders of magnitude smaller than the exact indices.

5.5.4 Comparison of Efficiency vs. Top-k

In this subsection, we compare the efficiency of the algorithms. We vary the query parameter k from 10 to 90, and show in Figure 5.3(b) the query execution time (log scale) of Empty, Horizon, Vertical, and Prob with respect to k. When $k \leq 30$, Horizon uses < 1 second and is the fastest.



(a) Storage overhead

(b) Runtime vs. k

(c) # probe queries vs. k

(d) # verified objects vs. k

Figure 5.3: Performance comparison w.r.t. top-k.

When $k > 30$, Prob is the fastest. We can see that Prob consistently outperforms Vertical by 100 to 1000 times, and beats Horizon by 10 to 100 times when $k > 30$. Overall, Prob achieves dramatically higher efficiency while using less space.

To explain the gap of the query execution time of these methods, Figure 5.3(c) plots the number of probe queries issued vs. top-k for each method. This curve is strongly correlated with Figure 5.3(b), which verifies that the probing cost dominates the total runtime. We can see that Empty needs to probe all 120 object spaces. Vertical, materializes $L' = 12$ sorted lists of scores, and therefore performs slightly better than Empty. Because each query object has descendent objects in all object spaces, Vertical's performance is invariant to the query. For Horizon, the number of probed spaces is monotonically increasing. In particular, 0 space is probed when $k \leq 30$ and 84 spaces are probed at $k = 90$. This verified that Horizon is not robust: it is *very sensitive* to the parameter k, because Horizon only materializes the highly ranked objects, thereby having no pruning power when the k-th rank threshold is not high. On the contrary, Prob performs much more consistently and is *insensitive* to the parameter k. In fact, Figure 5.3(c) shows that Prob probes no more than 2 object spaces in the worst case!

In Figure 5.3(d) we show the number of objects verified (*i.e.*, how many objects' true rank has been computed during the probing phase) in relation to k. On average, each query object has $|C(p)| = 25395.4$ descendent objects. For Vertical only about 110 of all the descendent objects are not verified. However, despite that the number of unverified objects is few in this case, some object spaces can be pruned and thus it is slightly faster than Empty. For Horizon, about half of the objects are not verified when $k \geq 60$. Not surprisingly, Prob has a dramatic improvement over the exact algorithms in that only very few objects are computed.

5.5.5 Pruning Power of the Probabilistic Approach

We further analyze the pruning power of Prob by plotting Figure 5.4. We vary k from 10 to 1000 in log scale and plot three curves: "Total" (the total number descendent objects), "Boundary" (the number of boundary objects), and "Verified" (the number of boundary objects left unpruned, *i.e.*, $|B^c|$). (Note that the "Verify" curve should have value 0 at $k = 20$ but is shown to be 0.1 in the figure.)

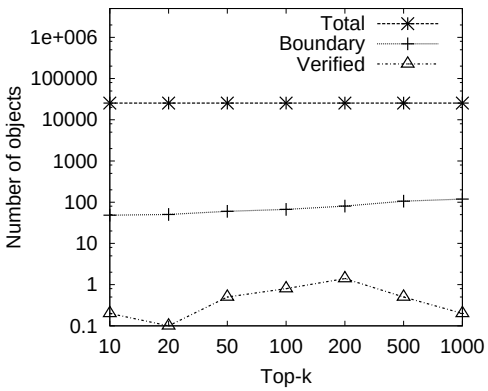


Figure 5.4: Probabilistic pruning power when θ is set to 0.8.

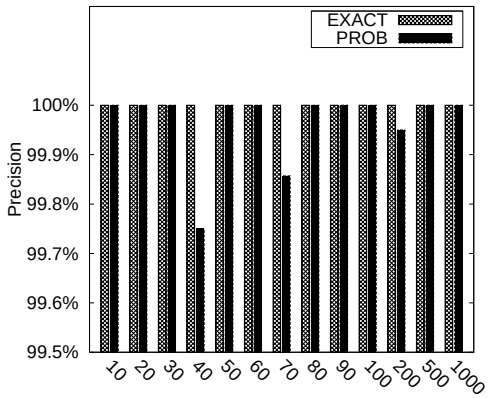


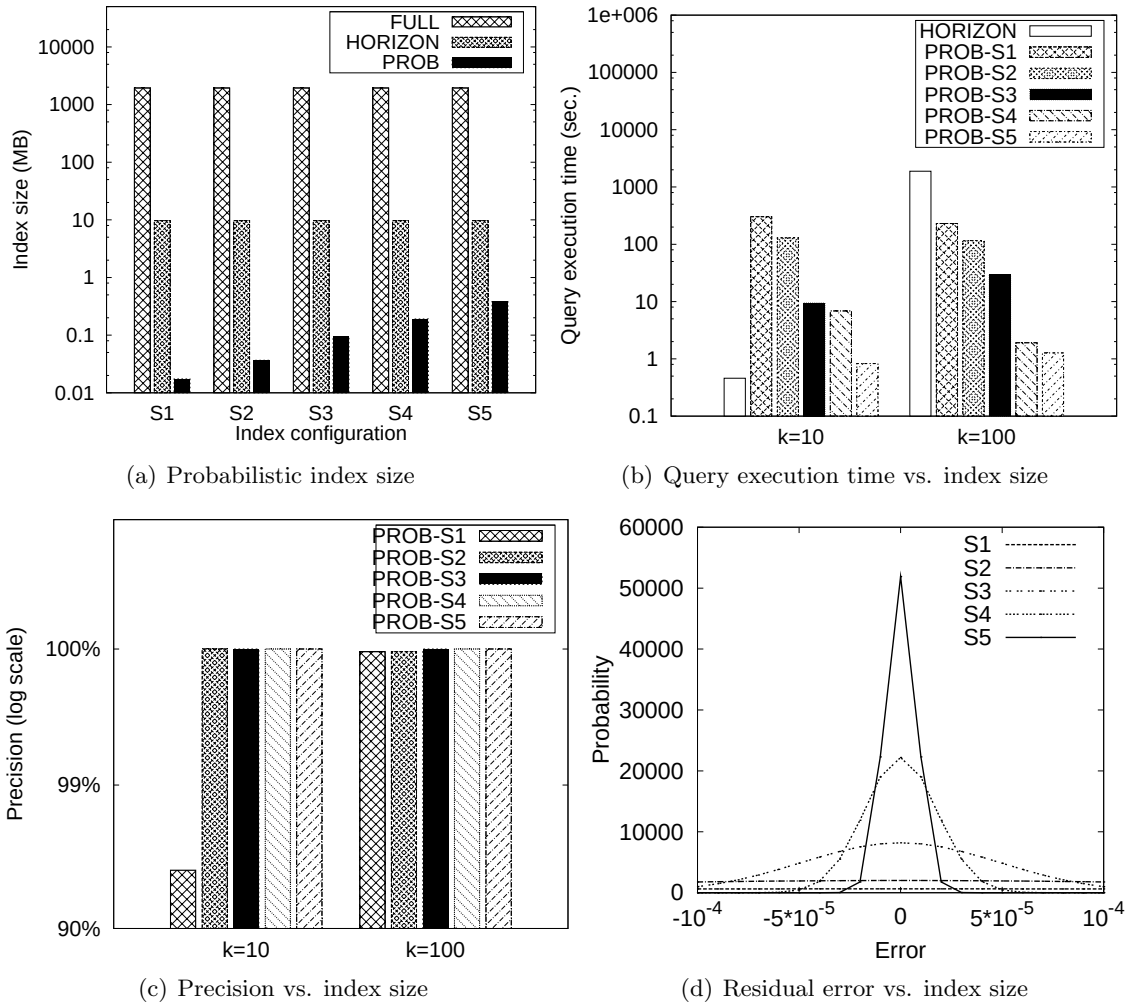
Figure 5.5: Precision of the probabilistic approach.

On average, we can see that "Boundary" is about 1/300 of "Total", and "Verified" is about 1/100. This figure confirms that the probabilistic pruning approach is extremely effective, where most objects can be pruned with high confidence. Unlike an exact algorithm like Horizon, whose performance degrades quickly as k becomes larger, Prob is very insensitive to k as the number of objects verified is constantly ≤ 1.5. Thus, such a probabilistic pruning approach is more desirable in real applications.

5.5.6 Precision Guarantee

For Prob, it is critical to ensure its result quality. We now turn to a precision evaluation for Prob. We keep $\theta = 0.8$ and vary k from 10 to 100, 200, 500, and 1000. The precision of Prob is plotted in Figure 5.5. Interestingly, despite that our confidence threshold is set to 0.8, we found that the precision of Prob is 100% for most k values. In particular, the precision is 99.75%, 99.86%, and 99.95% at $k = 40$, 70, and 200, respectively. In fact, for each of these k values, only 1 top-k object outputted by Prob is not accurate among all 10 workload queries.

Our conclusion here is twofold. First, Prob is able to yield a desirable tradeoff between efficiency and precision. Combining Figures 5.4 and 5.5, it is evident that a large number of boundary objects can be pruned with very little sacrifice in precision. In general, the precision is higher when k is larger because of the fact that the absolute error (*i.e.*, the number of non-top-k objects produced by Prob) is consistently low. Second, because the empirical precision is much higher than the theoretical threshold 0.8, the user-specified confidence θ is somewhat "conservative" than being



(a) Probabilistic index size

(b) Query execution time vs. index size

(c) Precision vs. index size

(d) Residual error vs. index size

Figure 5.6: Performance of psIndex w.r.t. index size.

"aggressive". This is desirable in any application with a need for quality assurance.

Since the precision of the results is also related to different values of θ and N (the number of samples), we will further study those parameters shortly.

5.5.7 Probabilistic Score Index Size

In this subsection, we evaluate Prob's performance when varying the space budget parameter α. We generate five indices, $S1$, $S2$, $S3$, $S4$, and $S5$, using $\alpha = 0.001\%$, 0.002%, 0.005%, 0.01%, and 0.02%, respectively. $S3$ is the default size configuration studied in Section 5.2.1. As shown in Figure 5.6(a), Prob's size is from $\frac{1}{500}$ to $\frac{1}{25}$ of Horizon's.

Figure 5.6(b) displays a comparison of query execution time between $S1$ to $S5$ with Horizon.

The comparison is divided into two groups corresponding to $k = 10$ and $k = 100$, respectively. As expected, Prob's efficiency is higher when increasing its size. At $k = 10$, Horizon has the optimal performance as all top-k answers are precomputed; on the other hand, $S5$ has very similar performance with Horizon even when its size is only $\frac{1}{25}$ of Horizon. At $k = 100$, all Prob configurations outperform Horizon, from 8 ($S1$) to 1488 times (S_5).

Figure 5.6(c) displays Prob's precision. For $S3$, $S4$, and $S5$, the precision is 100%. For $S2$, the precision is 100% at $k = 10$ and 99.9% at $k = 100$. For $S1$, the precision is 92% at $k = 10$ and 99.9% at $k = 100$. The precision is higher when the index is larger. Further, the precision is consistently high (*e.g.*, $\geq 99.9\%$) when the index size is reasonably large (*i.e.*, $\alpha \geq 0.002\%$).

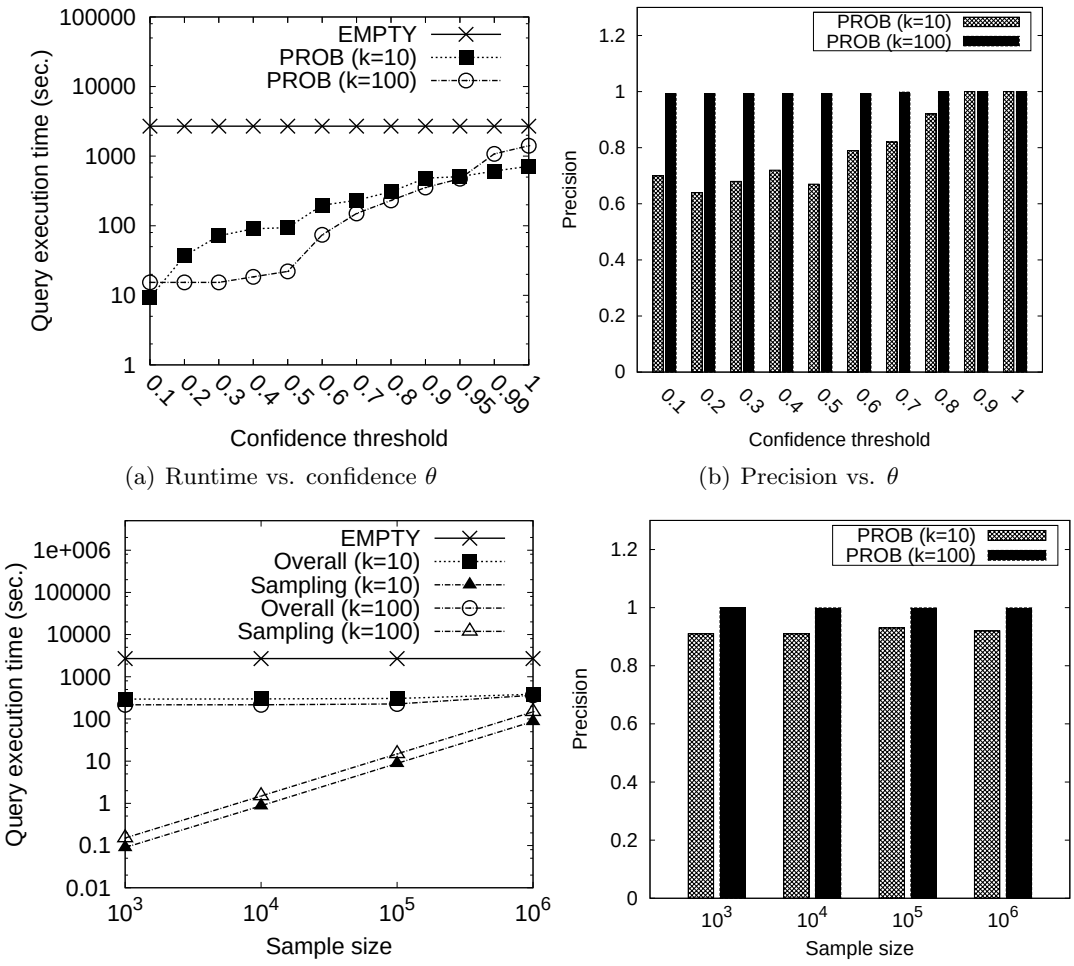
Figure 5.6(d) plots the residual error's probability density distributions for $S1$ through $S5$. This matches our intuition: when the index size becomes larger, the peak (centered at 0) is higher and the variance is smaller, indicating a better quality of regression.

5.5.8 Confidence Threshold

We evaluate the impact of the confidence threshold θ on Prob's performance in this subsection. We fix $\alpha = 0.005\%$ for Prob and vary θ from 0.0 to 1.0. Figure 5.7(a) displays the query execution time of Empty, Prob at $k = 10$, and Prob at $k = 100$ with respect to θ. We can see that a smaller value of θ would result in less query execution time. This is because smaller θ would cause a monotonically larger number of boundary objects to be pruned during the probabilistic pruning step (*i.e.*, larger B^+ and B^-).

Figure 5.7(b) shows the precision of query results with respect to θ. Observe that (1) the precision is consistently above the threshold θ; (2) The precision increases when θ increases; and (3) the precision is 100% when $\theta \geq 0.9$. These results empirically verified that θ is able to well represent the user's confidence level.

Figures 5.7(a) and 5.7(b) exhibit a controllable tradeoff between efficiency and precision using the parameter θ. While in general we found that a confidence threshold of 0.8 works well in practice, the optimal parameter of θ can be selected on an application-dependent basis.



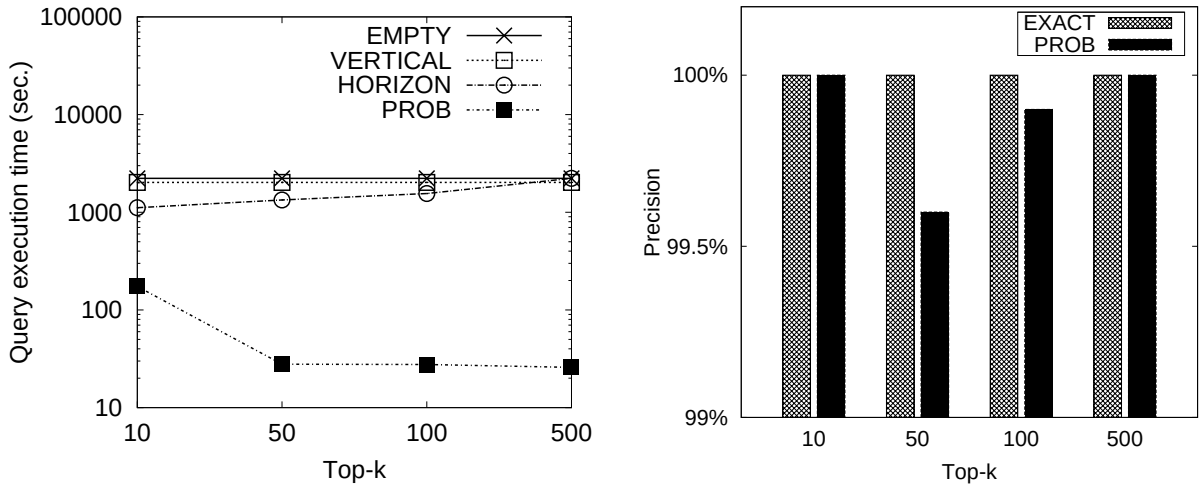
(a) Runtime vs. confidence θ

(b) Precision vs. θ

(c) Runtime vs. sample size N

(d) Precision vs. N

Figure 5.7: Performance of psIndex w.r.t. probabilistic parameters.



(a) Runtime vs. k

(b) Precision vs. k

Figure 5.8: The AVG aggregate function.

5.5.9 Random Sampling Size

During the random sampling step, a too small sample size N is likely to derive incorrect probability $\Pr(j)$ for a boundary object b_j, whereas a too large sample size may undermine the efficiency of Prob. How large is the random sampling size sufficient? To answer this question, we investigate the impact of the number of random samples (N) on Prob's performance.

To make N's impact more obvious, We fix $\theta = 0.8$ and set $\alpha = 0.001\%$ (corresponding to $S1$ in the index size experiment).

We vary N from 10^3 to 10^6. Figure 5.7(c) plots the relation between N and query execution time. We plot the overall time as well as the time exclusively used by the random sampling process (in log scale). We can see that, when $N = 10^3$, 10^4, and 10^5, the sampling process uses only $\frac{1}{1400}$, $\frac{1}{140}$, and $\frac{1}{15}$ of the overall time, indicating that sampling has a very small overhead.

Figure 5.7(d) displays the query result precision with respect to N. We observe that the precision is *extremely insensitive* to N. In fact, at $k = 10$, the precision is consistently about 92%, and at $k = 100$, the precision is consistently $\geq 99.9\%$.

We conclude from Figures 5.7(c) and 5.7(d) that a reasonably large N (*e.g.*, 10^5) introduces only a neglectable amount of overhead and is able to give good estimation for the probability computation.

5.5.10 Other Aggregate Functions

Finally, we study the performance of Prob on the AVG (average) aggregate function instead of SUM. As in the default case, we use $\alpha = 0.5\%$ for Horizon and Vertical and $\alpha = 0.005\%$ for Prob; we also set $\theta = 0.8$ and $N = 10^5$.

Figure 5.8(a) shows the query execution time with respect to k for Empty, Horizon, Vertical, and Prob. Horizon is the fastest exact algorithm in this figure. We vary k from 10 to 500. At $k = 10$, Prob is 6.3 times faster than Horizon. At $k = 50$, 100, and 500, Prob is 48 to 86 times faster than Horizon. Note that Prob's time does not necessarily increase when k is increased because k does not dictate the number of boundary objects.

Figure 5.8(b) plots the precision of Prob with respect to k. Prob has accuracy 100%, 99.6%, 99.9%, and 100% in these four cases, verifying that the quality of the regression and pruning steps is high regardless of the aggregate function used.

In summary, while relaxing very little on the accuracy, Prob is able to outperform the exact Horizon strategy in both storage overhead (2 orders of magnitude) and efficiency (1 to 2 orders of magnitude).

5.6 Extensions

5.6.1 Handling Complex Query Semantics

We discuss two extensions to the kOSearch query model: (1) how to support query-dependent weights on different object spaces (*e.g.*, an object that is ranked 1st in A may be considered more interesting than it ranked 1st in ABC); and (2) how to avoid redundant top-k results (*e.g.*, objects like a_2b_3 and $a_2b_3c_1$ may contain redundant information).

Given a set of non-negative weights $w_1, w_2, \ldots, w_L$ for the object spaces, the query execution algorithms 1 and 3 can be extended as follows. When computing the rank bounds $HRank$ and $LRank$ for each descendent object, we can incorporate these weights and derive a set of weighted bounds. The threshold computation and the pruning step remain unchanged. By letting $w_l = |S_l|$ $(1 \leq l \leq L)$, we can model the absolute ranking semantics. More complex weighting schemes can be in principle supported in a similar way. These weights would not significantly affect the

Problem context	Object space	Application
Multi-dimensional	*Group-by queries*	*Decision support*
Keyword search	*Keyword queries*	*Advertising*
General problem	*Ranked queries*	*Social networks, ...*

Table 5.7: kOSearch problem formulations where the probabilistic score index framework can be applied.

algorithm's performance.

To avoid redundancy, the algorithms can be modified as follows. Instead of generating the top-k objects as a *whole*, the original algorithms can generate the top-k results *one at a time*. First, a top object is generated as if $k = 1$. Second, all redundant objects with respect to the top-1 are ignored in the subsequent computation. This iterative method can remove the redundancy from the top-k results. Further, this method does not introduce significant efficiency overhead because any object space is guaranteed to be probed at most once.

5.6.2 Incremental Index Maintenance

In real data warehousing applications, object scores may not be static; they can be incrementally updated over time. This calls for incremental index maintenance techniques to accommodate such dynamic changes.

We discuss an extension to the probabilistic score index framework. Our extended solution contains three component. First, to dynamic maintain the knots in each object space, we can borrow existing algorithms like [62], which is able to maintain the knots within a guaranteed error bound using limited memory space. Thus, *HRank* and *LRank* can be accurately derived from the dynamically maintained knots. Second, the regression parameters can be updated by building a predictive model over time. The residual error, on the other hand, can also be estimated using random sampling techniques. Third, a background thread will be constantly running that computes the up-to-date regression models for each object space in an iterative fashion. This thread will not block query processing and therefore will not affect the performance.

5.6.3 The General kOSearch Query Problem and Its Applications

In this chapter, we studied the kOSearch queries defined over the multi-dimensional space, where the object spaces are defined as multi-dimensional group-by's. However, the kOSearch query model can generalize a broad class of problems, as illustrated in Table 5.7. The probabilistic score index framework developed here can be applied toward all these problems.

For example, in the keyword search domain, object spaces may be formulated by *keyword queries*, and objects represent *documents* or other entities. Given an object DOC, a user may like to discover the most interesting keyword queries $KWQ_1, KWQ_2, \ldots, KWQ_k$ such that DOC is highly ranked in the result list of each KWQ_i $(1 \leq i \leq k)$. In a typical application, a keyword query generates a list of result documents ranked by a relevance scoring function. An application of such a kOSearch problem is the *search-based advertising*: advertisers may find it useful to discover the most interesting keyword queries to promote their webpages.

The kOSearch queries can be generally formulated over other types of ranked queries with numerical scores. For example, in a *social network*, one may like to find the best subnetwork of links such that a user or product is highly ranked by centrality or network distance measures. We leave a more in-depth examination of kOSearch queries on these application domains and extensions of the probabilistic score index framework to our future work.

5.7 Summary

We have proposed the rank-driven top-k object search problem to discover the "best" sub-products or features for a given product. We studied both exact and approximate strategies to seek a middle ground between time and space. The probabilistic score index framework was developed for top-k approximation with probabilistic guarantee. Comprehensive experiments have verified the effectiveness and efficiency of our methods.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis motivates the general promotion analysis research problem. The existing online analytical processing and multidimensional analysis engines suffer from a lack of effective and efficient support for promotion-oriented functionalities. Despite the need for such functionalities is ubiquitous in a wide variety of real-world applications, users with a traditional decision support system must go through heavy guess work to discover interesting spaces for promotion, and sometimes it would be simply impossible for them to obtain any interesting results.

To address the needs, a systematic, principled promotion analysis framework was developed in this thesis. We discussed the models and semantics of the promotion analysis problem. A unified interestingness measure combining the object rank and rank-independent properties of local spaces was proposed to model various semantics. More advanced redundancy-aware query model was also examined. Furthermore, we discussed both categorical and continuous, ranged dimensions for the data space, as well as flat and multidimensional domains for the object space.

The promotion analysis problem brought many technical challenges. For example, we tackled the spurious promotion problem and the expensive holistic property of object ranking. Also, we developed efficient techniques for object aggregation and ranking as well as an optimal algorithm to achieve the best tradeoff between space and time. We explored both exact and approximate query optimization strategies, which were shown to work well empirically. In summary, our framework has the following benefits:

- Flexibility: It allows users to conduct promotion analysis on various data contexts and specify different query semantics;

- Efficiency: Our proposed approaches will be able to efficiently answer promotion queries and enable materialization plans that yield a desirable tradeoff between space overhead and efficiency;
- Scalability: All of our proposed approaches are scalable to deal with the size of large data warehouses;
- Practicality: Our proposed approaches do not rely on unrealistic assumptions and may be implemented in existing systems conveniently;
- Extensibility: Our proposed approaches demonstrate general principles that may be applied toward other problems.

Based on the above discussion, we believe that the promotion analysis framework is an interesting and important study.

6.2 Future Directions

There could be various ways to model promotion applications. Here we discuss several potentially interesting themes for future research.

A language for promotion analysis: In this thesis, the multidimensional data space and object space are predefined. In practice, users may dynamically construct these spaces or provide fuzzy guidance for promotion search. For example, when promoting an author in the DBLP data set, the user may define some research areas according to her own criteria, or provide hints to guide the promotion process. Therefore, it is important to develop a formal language for users to flexibly conduct the analysis. The key challenge would be to abstract out a set of basic primitives that is expressive and complete. For online computation, primitive-specific optimization can be performed to make the analysis efficient.

Feature discovery and selection: When the search space for promotion is undefined or unknown, it is unclear what features one may use to promote an object. In order to discover interesting features and judge the goodness of each feature, machine learning techniques may be developed. For example, we can use supervised or semi-supervised techniques to learn what product

properties are meaningful for promotion, and what types of ranked results are truly useful yet surprising. Further exploration in this topic would complement our study and make the framework more powerful.

Group promotion: Instead of promoting a single target object as discussed in the previous chapters, users may want to promote a group of target objects. For example, the target group could be a collection of different types of items in a product package (*e.g.*, a travel package including hotel and flight). For such group promotion, the promotiveness can be measured by the average increase in rank for each object in the target group or by other user-defined aggregate measures. How to extend the cost model and approximate techniques, on the other hand, would be an interesting open question.

Mining the measure space: Mining promotive regions in the measure space is an orthogonal yet important problem. Unlike the multidimensional space that is organized in a finite number of cells, the measure space is often numerical and thus impossible to be exhaustively enumerated. On the other hand, it would be very useful for decision makers to understand the product attributes and position products. For example, knowing by what criteria a product is successful (*e.g.*, by sales or by customer rating) can help further position and promote it. Thus, an in-depth examination on how to enable the interaction between promotion analysis and the numerical space of measures and aggregates would be worthwhile.

This thesis work may also open up new horizons in other domains such as large heterogeneous information network analysis, graph mining, and spatio-temporal data mining. Below we outline several domains.

Promotion in information networks: A heterogeneous information network, with each node and link carrying some multidimensional information, may need promotion analysis as well to promote objects in such a network or in its surrounding subnetworks. However, the promotion measures of an object could be related to certain network property, such as network density, connectivity, and centrality; the computation of such a promotion measure could be closely related to the network topological structure and the node/link values. The methods for promotion analysis will need to be re-examined in such networks. We propose to perform network-based precomputation so that an initial evaluation of the nodes and links can be done before query time such that

the online query-based promotion computation can avoid searching many hopeless paths.

An example could be the LinkedIn network (http://www.linkedin.com/), where nodes represent professional persons and two persons have an edge when they are connected in the real world. In a traditional scenario of search, one may issue a boolean query and perform aggregations over the network to obtain the top-k nodes with the highest scores (*e.g.*, find the top-k persons connected with the most "database" professionals in a 2-hop neighborhood). From a promotional perspective, the user may want to discover interesting boolean queries which can promote a given node or community (*e.g.*, given a person, find in which area he is highly ranked in professional connections). The detailed search strategy is left for further work.

Ranking and promotion over textual data: One can extend the promotion analysis framework to deal with unstructured data and queries in addition to the structured dimensions. It would be interesting to discover useful keywords for promotion or, conversely, conduct promotion with respect to an input keyword query. Relevance ranking or other information retrieval metrics may be applied to gauge the utility of results. Moreover, one can study the problem of comparative ranking/search that involves not only a query object for analysis, but also a given pool of competitor objects for comparison. These problems will need more in-depth research in data mining and machine learning.

Probabilistic and uncertain data analysis: Recently, uncertain data management have been gaining increasing attention. The key difference between uncertain and certain data management is that the former enables users to model information uncertainty by associating tuples with probability values and correlation relationships. There has been a number of studies on probabilistic ranked query processing. We can further examine the promotion analysis problems in this area. For instance, given a target object, users may like to know which subspaces or regions an object has the highest expected rank or the largest probability to be highly ranked. These questions pose many research challenges that have not been addressed before.

References

[1] DBLP. http://www.informatik.uni-trier.de/~ley/db/.

[2] NBA data set, http://www.basketballreference.com.

[3] TPC-H. http://www.tpc.org/tpch/.

[4] Chidanand Apté, Bing Liu, Edwin P. D. Pednault, and Padhraic Smyth. Business applications of data mining. *Commun. ACM*, 45(8):49–53, 2002.

[5] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 475–486, Seoul, Korea, Sep. 2006.

[6] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *The VLDB Journal*, 15(3):229–249, 2006.

[7] Michael J. A. Berry and Gordon Linoff. *Data Mining Techniques: For Marketing, Sales, and Customer Support*. Wiley, May 1997.

[8] Kevin S. Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 359–370, Philadelphia, PA, USA, Jun. 1999.

[9] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In *Proc. of the 23rd International Conference on Data Engineering (ICDE) Workshops*, pages 506–515, Istanbul, Turkey, Apr. 2007.

[10] Christian Borgs, Jennifer T. Chayes, Nicole Immorlica, Kamal Jain, Omid Etesami, and Mohammad Mahdian. Dynamics of bid optimization in online advertisement auctions. In *Proc. of the 16th International Conference on World Wide Web (WWW)*, pages 531–540, Banff, Alberta, Canada, May 2007.

[11] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proc. of the 17th International Conference on Data Engineering (ICDE)*, pages 421–430, Heidelberg, Germany, Apr. 2001.

[12] Kaushik Chakrabarti, Venkatesh Ganti, Jiawei Han, and Dong Xin. Ranking objects based on relationships. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 371–382, Chicago, Illinois, USA, Jun. 2006.

[13] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Jun. 2002.

[14] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[15] Moses Charikar and Rina Panigrahy. Clustering to minimize the sum of cluster diameters. In *Proc. on 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–10, Heraklion, Crete, Greece, Jul. 2001.

[16] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Transactions on Database Systems*, 31(3):1134–1168, 2006.

[17] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, 1997.

[18] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip S. Yu. Graph olap: a multi-dimensional framework for graph data analysis. *Knowledge and Information Systems*, 21(1):41–63, 2009.

[19] Yixin Chen, Guozhu Dong, Jiawei Han, Jian Pei, Benjamin W. Wah, and Jianyong Wang. Regression cubes with lossless compression and aggregation. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1585–1599, 2006.

[20] Chun Kit Chui, Eric Lo, Ben Kao, and Wai-Shing Ho. Supporting ranking pattern-based aggregate queries in sequence data cubes. In *Proc. of the 18th ACM Conference on Information and Knowledge Management (CIKM)*, pages 997–1006, Hong Kong, China, Nov. 2009.

[21] Graham Cormode, Minos N. Garofalakis, S. Muthukrishnan, and Rajeev Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 25–36, Baltimore, Maryland, USA, Jun. 2005.

[22] Thomas Cover and Joy Thomas. *Elements of Information Theory 2nd Edition*. Wiley-Interscience, 2006.

[23] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Nikos Sarkas. Ad-hoc top-k query answering for data streams. In *Proc. of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 183–194, Vienna, Austria, Sep. 2007.

[24] Gautam Das, Vagelis Hristidis, Nishant Kapoor, and S. Sudarshan. Ordering the attributes of query results. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 395–406, Chicago, Illinois, USA, Jun. 2006.

[25] Bolin Ding, Bo Zhao, Cindy Lin, Jiawei Han, and ChengXiang Zhai. Topcells: Keyword-based search of top-k aggregated documents in text cube. In *International Conference on Data Engineering (ICDE)*, Long Beach, California, USA, Mar. 2010.

[26] Srinivas Doddi, Madhav V. Marathe, S. S. Ravi, David Scot Taylor, and Peter Widmayer. Approximation algorithms for clustering to minimize the sum of diameters. In *7th Scandinavian Workshop on Algorithm Theory*, pages 237–250, Bergen, Norway, Jul. 2000.

[27] Guozhu Dong, Jiawei Han, Joyce M. W. Lam, Jian Pei, and Ke Wang. Mining multi-dimensional constrained gradients in data cubes. In *Proc. of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 321–330, Rome, Italy, Sep. 2001.

[28] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. In *Proc. of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 28–36, San Francisco, CA, USA, Jan. 2003.

[29] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proc. of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Santa Barbara, California, USA, May 2001.

[30] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *Proc. of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 299–310, New York City, USA, Aug. 1998.

[31] Matt Gibson, Gaurav Kanade, Erik Krohn, Imran A. Pirwani, and Kasturi R. Varadarajan. On clustering to minimize the sum of radii. In *Proc. of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 819–825, San Francisco, California, USA, Jan. 2008.

[32] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 454–465, Hong Kong, China, Aug. 2002.

[33] Hector Gonzalez, Jiawei Han, and Xiaolei Li. Flowcube: Constructuing rfid flowcubes for multi-dimensional analysis of commodity flows. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 834–845, Seoul, Korea, Sep. 2006.

[34] Amit Goyal, Francesco Bonchi, and Laks V. S. Lakshmanan. Discovering leaders from community actions. In *Proc. of the 17th ACM Conference on Information and Knowledge Management (CIKM)*, pages 499–508, Napa Valley, California, USA, Oct. 2008.

[35] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Mining and Knowledge Discovery (DMKD)*, 1(1):29–53, 1997.

[36] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 58–66, Santa Barbara, California, USA, May 2001.

[37] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques, 2nd ed.* Morgan Kaufmann, March 2006.

[38] Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–12, Santa Barbara, California, USA, May 2001.

[39] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 205–216, Montreal, Canada, Jun. 1996. ACM Press.

[40] Jason D. Hartline, Vahab S. Mirrokni, and Mukund Sundararajan. Optimal marketing strategies over social networks. In *Proc. of the 17th International Conference on World Wide Web*, pages 189–198, Beijing, China, Apr. 2008.

[41] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range queries in olap data cubes. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 73–88, Tucson, Arizona, USA, May. 1997.

[42] Dorit S. Hochbaum. *Approximation algorithms for NP-hard problems*. PWS Pub. Co., 1997.

[43] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 850–861, Berlin, Germany, Sep. 2003.

[44] Ming Hua, Jian Pei, Wenjie Zhang, and Xuemin Lin. Ranking queries on uncertain data: a probabilistic threshold approach. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 673–686, Vancouver, Canada, Jun. 2008.

[45] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal*, 13(3):207–221, 2004.

[46] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top- query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4), 2008.

[47] Kamal Jain and Vijay V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *J. ACM*, 48(2):274–296, 2001.

[48] Jon M. Kleinberg, Christos H. Papadimitriou, and Prabhakar Raghavan. A microeconomic view of data mining. *Data Mining and Knowledge Discovery (DMKD)*, 2(4):311–324, 1998.

[49] Philip Kotler and Kevin Keller. *Marketing Management*. Prentice Hall, March 2008.

[50] S Kotsiantis and D Kanellopoulos. Association rules mining: A recent overview. *International Transactions on Computer Science and Engineering*, 32(1):71–82, 2006.

[51] Robert O. Kuehl. *Design of Experiments: Statistical Principles of Research Design and Analysis*. Duxbury/Thomson Learning, 2000.

[52] Peter Lancaster and Kestutis Salkauskas. *Curve and surface fitting. An introduction*. Academic Press, Feb. 1986.

[53] Cuiping Li, Beng Chin Ooi, Anthony K. H. Tung, and Shan Wang. Dada: a data cube for dominant relationship analysis. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 659–670, Chicago, Illinois, USA, Jun. 2006.

[54] Jian Li, Barna Saha, and Amol Deshpande. A unified approach to ranking in probabilistic databases. *Proc. of the VLDB Endowment*, 2(1):502–513, 2009.

[55] Xiaolei Li and Jiawei Han. Mining approximate top-k subspace anomalies in multi-dimensional time-series data. In *Proc. of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 447–458, Vienna, Austria, Sep. 2007.

[56] Xiaolei Li, Jiawei Han, and Hector Gonzalez. High-dimensional olap: A minimal cubing approach. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 528–539, Toronto, Ontario, Canada, Sep. 2004.

[57] Xiang Lian and Lei Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 213–226, Vancouver, Canada, Jun. 2008.

[58] Cindy Xide Lin, Bolin Ding, Jiawei Han, Feida Zhu, and Bo Zhao. Text cube: Computing ir measures for multidimensional text database analysis. In *Proc. of the 8th IEEE International Conference on Data Mining (ICDM)*, pages 905–910, Pisa, Italy, Dec. 2008.

[59] Eric Lo, Ben Kao, Wai-Shing Ho, Sau Dan Lee, Chun Kit Chui, and David W. Cheung. Olap on sequence data. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 649–660, Vancouver, Canada, Jun. 2008.

[60] Yi Luo, Xuemin Lin, Wei Wang 0011, and Xiaofang Zhou. Spark: top-k keyword query in relational databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 115–126, Beijing, China, Jun. 2007.

[61] Hao Ma, Haixuan Yang, Michael R. Lyu, and Irwin King. Mining social networks using heat diffusion processes for marketing candidates selection. In *Proc. of the 17th ACM Conference on Information and Knowledge Management (CIKM)*, pages 233–242, Napa Valley, California, USA, Oct. 2008.

[62] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 426–435, Seattle, WA, USA, Jun. 1998.

[63] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 251–262, Philadelphia, PA, USA, Jun. 1999.

[64] Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating top-k queries over web-accessible databases. *ACM Transactions on Database Systems*, 29(2):319–362, 2004.

[65] Muhammed Miah, Gautam Das, Vagelis Hristidis, and Heikki Mannila. Standing out in a crowd: Selecting attributes for maximum visibility. In *Proc. of the 24th International Conference on Data Engineering (ICDE)*, pages 356–365, Cancun, Mexico, Apr. 2008.

[66] Chihiro Ono, Mori Kurokawa, Yoichi Motomura, and Hideki Asoh. A context-aware movie preference model using a bayesian network for recommendation and promotion. In *User Modeling*, pages 247–257, 2007.

[67] Athanasios Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, 1991.

[68] Jian Pei, Wen Jin, Martin Ester, and Yufei Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *Proc. of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 253–264, Trondheim, Norway, Aug. 2005.

[69] Raghu Ramakrishnan and Bee-Chung Chen. Exploratory mining in cube space. *Data Mining and Knowledge Discovery (DMKD)*, 15(1):29–54, 2007.

[70] James B. Ramsey. Tests for specification errors in classical linear least-squares regression analysis. *Journal of the Royal Statistical Society. Series B (Methodological)*, 31(2), 1969.

[71] Christopher Re, Nilesh N. Dalvi, and Dan Suciu. Efficient top-k query evaluation on probabilistic data. In *Proc. of the 23rd International Conference on Data Engineering (ICDE)*, pages 886–895, Istanbul, Turkey, Apr. 2007.

[72] Matthew Richardson and Pedro Domingos. Mining knowledge-sharing sites for viral marketing. In *Proc. of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 61–70, Edmonton, Alberta, Canada, Jul. 2002.

[73] Paat Rusmevichientong and David P. Williamson. An adaptive algorithm for selecting profitable keywords for search-based advertising services. In *Proc. 7th ACM Conference on Electronic Commerce*, pages 260–269, Ann Arbor, Michigan, USA, Jun. 2006.

[74] Jayavel Shanmugasundaram, Usama M. Fayyad, and Paul S. Bradley. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In *Proc. of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 223–232, San Diego, California, USA, Aug. 1999.

[75] Michael J. Shaw, Chandrasekar Subramaniam, Gek Woo Tan, and Michael Welge. Knowledge management and data mining for marketing. *Decision Support Systems*, 31(1):127–137, 2001.

[76] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. Probabilistic top- and ranking-aggregate queries. *ACM Transactions on Database Systems (TODS)*, 33(3), 2008.

[77] Yufei Tao, Ke Yi, Sheng Cheng, Jian Pei, and Feifei Li. Logging every footstep: Quantile summaries for the entire history. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Indianapolis, Indiana, USA, Jun. 2010.

[78] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 648–659, Toronto, Ontario, Canada, Sep. 2004.

[79] Jeffrey Scott Vitter and Min Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 193–204, Philadelphia, PA, USA, Jun. 1999.

[80] Jeffrey Scott Vitter, Min Wang, and Balakrishna R. Iyer. Data cube approximation and histograms via wavelets. In *Proc. of the 1998 ACM CIKM International Conference on Information and Knowledge Management (CIKM)*, pages 96–104, Bethesda, Maryland, USA, Nov. 1998.

[81] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Norvag. Reverse top-k queries. In *International Conference on Data Engineering (ICDE)*, Long Beach, California, USA, Mar. 2010.

[82] Qian Wan, Raymond Chi-Wing Wong, Ihab F. Ilyas, M. Tamer Özsu, and Yu Peng. Creating competitive products. *Proc. of the VLDB Endowment*, 2(1):898–909, 2009.

[83] Tianyi Wu, Yuguo Chen, and Jiawei Han. Association mining in large databases: A re-examination of its measures. In *Proc. of the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 621–628, Warsaw, Poland, Sep. 2007.

[84] Tianyi Wu, Yuguo Chen, and Jiawei Han. Re-examination of interestingness measures in pattern mining: A unified framework. *Data Mining and Knowledge Discovery (DMKD)*, 2010 (in print). (online pub. Jan. 06, 2010: DOI 10.1007/s10618-009-0161-2).

[85] Tianyi Wu, Xiaolei Li, Dong Xin, Jiawei Han, Jacob Lee, and Ricardo Redder. Datascope: Viewing database contents in google maps' way. In *Proc. of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 1314–1317, Vienna, Austria, Sep. 2007.

[86] Tianyi Wu, Yizhou Sun, Cuiping Li, and Jiawei Han. Region-based online promotion analysis. In *Proc. of the International Conference on Extending Data Base Technology (EDBT)*, pages 63–74, Lausanne, Switzerland, Mar. 2010.

[87] Tianyi Wu, Dong Xin, and Jiawei Han. Arcube: supporting ranking aggregate queries in partially materialized data cubes. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 79–92, 2008.

[88] Tianyi Wu, Dong Xin, Qiaozhu Mei, and Jiawei Han. Promotion analysis in multi-dimensional space. *Proc. of the VLDB Endowment*, 2(1):109–120, 2009.

[89] Dong Xin and Jiawei Han. Integrating olap and ranking: The ranking-cube methodology. In *Proc. of the 23rd International Conference on Data Engineering (ICDE) Workshops*, pages 253–256, Istanbul, Turkey, Apr. 2007.

[90] Dong Xin, Jiawei Han, Hong Cheng, and Xiaolei Li. Answering top-k queries with multi-dimensional selections: The ranking cube approach. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 463–475, Seoul, Korea, Sep. 2006.

[91] Xifeng Yan, Bin He, Feida Zhu, and Jiawei Han. Topk aggregation queries over large networks. In *International Conference on Data Engineering (ICDE)*, Long Beach, California, USA, Mar. 2010.

[92] Duo Zhang, ChengXiang Zhai, and Jiawei Han. Topic cube: Topic modeling for olap on multidimensional text databases. In *Proc. of the SIAM International Conference on Data Mining (SDM)*, pages 1123–1134, Sparks, Nevada, USA, Apr. 2009.

[93] Bin Zhou, Daxin Jiang, Jian Pei, and Hang Li. Olap on search logs: an infrastructure supporting data-driven applications in search engines. In *Proc. of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1395–1404, Paris, France, 2009.

Author's Biography

Tianyi Wu was born in Hunan, China. He attended the Fudan University, Shanghai, China, where he earned his Bachelor of Science degree in Computer Science in 2005. Following that, he entered the Computer Science Department of the University of Illinois at Urbana-Champaign (UIUC) and obtained his Master of Science degree in 2007. He received his Ph.D. from UIUC in 2010 under the supervision of Professor Jiawei Han.